مقرر: نظم الزمن الحقيقي

الفصل الأول: مفاهيم نظم الزمن الحقيقي

فهرس الفصل:

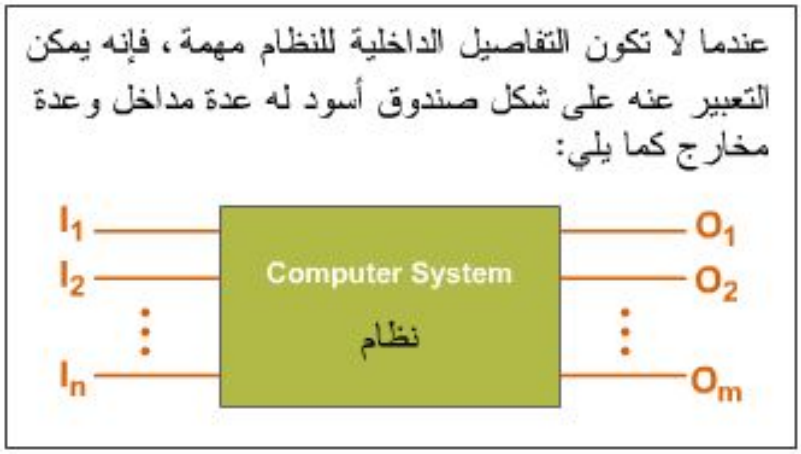
سنقدم في هذا الفصل المفاهيم الأساسية والمصطلحات اللازمة لفهم ماهية نظم الزمن الحقيقي. وسنقدم كذلك بعض الأمثلة الواقعية عنها ولمحة تاريخية عن تطورها.

1. المصطلحات الأساسية في نظام الزمن الحقيقي

مفهوم النظام System

يمكن تعريف النظام على أنه تقابل بين مجموعة مداخل **inputs** ومخارج **outputs**. ويمكن نمذجة أي كينونة في الطبيعة على شكل نظام.

عندما لا تكون التفاصيل الداخلية للنظام مهمة، فإنه يمكن التعبير عنه على شكل صندوق أسود له عدة مداخل وعدة مخارج كما يلي:

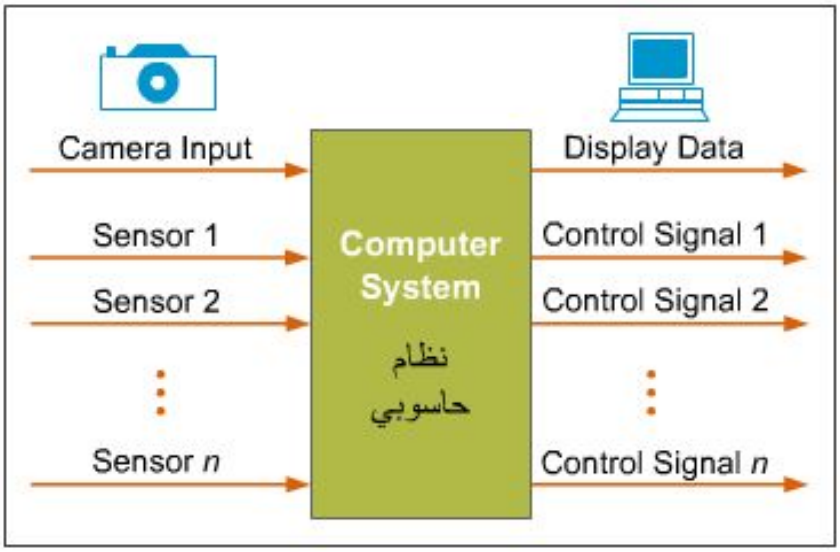


النظام الحاسوبية **computer systems** هي نظام مكونة من:

- **عتاد مادي hardware:** يشمل جميع الأجهزة الإلكترونية والكهربائية والميكانيكية في الحاسب، كالمعالج والذاكرة الرئيسية وأجهزة الدخل/خرج.
- **برمجيات software:** وهي مجموعة تعليمات ينفذها العتاد المادي (المعالج) لحل مسألة ما، وتقسم إلى نوعين:
  - **التطبيقات البرمجية software applications:** وهي مصممة لحل مشاكل محددة تهم المستخدم مثل حسابات الرواتب وإدارة المستودعات وتصفح الويب والألعاب وغيرها ...

- **برمجيات النظام system software:** التي تغلّف العتاد المادي وتؤمن للتطبيقات البرمجية واجهة تخاطب مجردة تمكنها من الوصول إلى خدمات العتاد المادي بسهولة. وغالباً ما تجمع برمجيات النظام الأساسية مثل مجدول المهام **scheduler** وسواقات الأجهزة **device drivers** ومدير الذاكرة **memory manager** في كتلة واحدة هي نظام التشغيل **operating system**.

تمثّل المداخل في النظم الحاسوبية أقنية المعلومات الرقمية[1] الواردة من الأجهزة الحاسوبية والبرمجيات الأخرى. أما مخارج النظم الحاسوبية فهي على الأغلب رقمية وتحول إلى تمثيلية لإرسالها إلى أجهزة الخرج المختلفة كالشاشات والسماعات وغيرها كما هو مبين في الشكل التالي:



وبالنظر إلى هذا النظام (الشكل)، نجد أن هناك تأخير زمني بين لحظة ورود المداخل (وتسمى أيضاً المحرضات **stimulus**) ولحظة ظهور المخارج الموافقة لها (وتسمى أيضاً الإجابة **response**).

إن زمن استجابة النظام: هو الزمن الفاصل بين لحظة ورود مداخل النظام ولحظة ظهور قيم الخرج على مخارجه.

---

[1] وهي تأتي غالباً من حساسات sensors كالكاميرات والميكروفونات وغيرها من الأجهزة التي تولد معطيـات رقمية digital أو تمثيلية analog.

نظام الزمن الحقيقي Real-time Systems

هو نظام يجب أن يحقق قيوداً صريحة (حدوداً محددة) على زمن الاستجابة. ويمكن أن يؤدي عدم احترام هذه القيود لنتائج خطيرة، أو حتى لفشل **failure** النظام بالكامل.

ماذا يعني "**فشل النظام**"؟ في بعض الحالات الخطيرة مثل فشل مكوك فضاء أو مفاعل نووي، يظهر فشل النظام واضحاً للعيان بنتائج كارثية مؤلمة. لكن في حالات أخرى مثل آلات الصراف الآلية **Automatic teller machines**، لا يظهر مفهوم الفشل بوضوح.

الفشل failure: هو عدم تمكن النظام من أداء مهمته حسب المواصفات المحددة له.

والنظام الفاشل failed system: هو نظام لا يمكنه أن يحقق أحد أو بعض المتطلبات المذكورة في دفتر شروطه.

توجد تعاريف أخرى لنظم الزمن الحقيقي تتفق كلها بضرورة تحقيق النظام لقيود زمنية لكي يكون صحيحاً. فعلى سبيل المثال، يمكن تعريف نظام الزمن الحقيقي كما يلي:

هو نظام تعتمد صحته منطقياً على صحة مخارجه بالإضافة لاحترامها لقيودها الزمنية.

ويمكن تقسيم نظم الزمن الحقيقي إلى:

- **نظم استجابة لأحداث Reactive:**

تستجيب لأحداث آتية من البيئة المحيطة بها، مثل نظم مراقبة الحريق التي تستجيب لارتفاع درجة الحرارة بتشغيل أجهزة الإنذار والإطفاء.

- **نظم مضمنة Embedded:**

وهي نظم تحكّم توجد في أنظمة أخرى ليست حاسوبية بالضرورة. على سبيل المثال:

- يوجد في معظم السيارات الحديثة نظم حاسوبية مضمنة تتحكم بحقن الوقود والكوابح وأكياس الهواء **airbag** وغيرها ...

تحوي العديد من الأجهزة المنزلية الحديثة على نظم مضمنة مثل أجهزة التلفاز والغسالات الآلية وأجهزة الستيريو.

ومن البديهي أيضاً أن النظم المعقدة كالطائرات وسفن الفضاء والآلات الصناعية تحوي كلها على نظم مضمنة. ينطبق شرط الزمن الحقيقي بشكلٍ خاص على هذه النظم.

ففي الطائرات مثلاً، يجب معالجة المعطيات الآتية من مقياس التسارع بدور محدد يعتمد على مواصفات الطائرة (كل **10** ميلي ثانية على سبيل المثال). وإذا فشل النظام في ذلك، فسيحصل

على معلومات خاطئة عن موقع وسرعة الطائرة، مما يمكن أن يؤدي إلى انحرافها عن مسارها في أحسن الأحوال، أو تحطمها في أسوأ الأحوال.

أما في المفاعلات النووية، فتؤدي عدم الاستجابة السريعة للمشاكل الحرارية إلى تعطله أو انفجاره في أسوأ الأحوال. أخيراً، يجب على نظام حجز التذاكر في شركة طيران الاستجابة لطلبات الزبائن خلال زمن معقول من وجهة نظر الزبائن، وإلا اعتبر نظاماً غير عملي.

باختصار، ليس من الضروري أن يستجيب نظام الزمن الحقيقي خلال زمنٍ قصير، يجب فقط أن يكون له قيود زمنية محددة (أجزاء من الثانية أو عدة ساعات) يستجيب خلالها.

- **نظام الزمن الحقيقي الرخو soft real-time system:** هو نظام تنقص أهمية الأعمال التي يؤديها إذا تعدى حدوده الزمنية، لكن لا تصبح عدية القيمة تماماً.

- **نظام الزمن الحقيقي الصعب hard real-time system:** هو نظام يؤدي تعديه لحدوده الزمنية لنتائج كارثية وتوقف كامل للنظام، حتى ولو حدث ذلك لمرة واحدة.

تجدر الإشارة إلى أن الحدود الفاصلة بين هذه الأنواع الثلاثة ليست قاطعة تماماً ويمكن لنظامٍ أن ينتقل من تصنيف لآخر في ظروفٍ خاصة.

على سبيل المثال، إذا تعدى جهاز الصراف الآلي حدوده الزمنية لعدد كبير من المرات، سيؤدي ذلك إلى درجة عالية من عدم الرضا لدى الكثير من زبائن المصرف وبالتالي تنخفض أرباح المصرف لخسارته لزبائنه مما يمكن أن يؤدي إلى إفلاسه بالكامل في أسوأ الأحوال. هذا يدل على أنه يمكن أن يتغير تصنيف نظام زمن حقيقي حسب الظروف الموضوع فيها.

- **نظام الزمن الحقيقي الفاسي firm real-time system:** هو نظام لا يؤدي تعديه لحدوده الزمنية لنتائج كارثية إذا حدث ذلك لعدد قليل من المرات. لكن إذا حصل ذلك لعدد أكبر من المرات، سيؤدي لنتائج كارثية وتعطل كامل للنظام.

في الحقيقة، يجب أن يصنّف أي نظام عملي وفق أحد هذه الأصناف الثلاثة[2]. وكأمثلة عن هذه النظم:

النظام	تصنيف نظام الزمن الحقيقي	الشرح
أجهزة الصراف الآلي **ATM**.	رخو	تعدي النظام لحدوده الزمنية لا يؤدي لنتائج كارثية بل يؤدي فقط لأداء متناقص للنظام.
نظام التحكم المضمن بآلة مؤتمتة لإزالة الأعشاب الضارة.	قاسي	تعدي النظام لحدود القيادة الزمنية يجعل الآلة تخرج عن السيطرة وتخرب المحاصيل المفيدة بدلاً من الأعشاب الضارة.
نظام التحكم بأسلحة طائرة حيث يؤدي الضغط على زر لإطلاق صاروخ جو-جو.	صعب	يمكن أن يؤدي عدم إطلاق الصاروخ ضمن الحد الزمني المحدد بعد الضغط على الزر إلى عدم إصابة الهدف وحدوث نتائج كارثية.

من المهم عند دراسة نظم الزمن الحقيقي أن نناقش ماهية الزمن، لأن حدودها الزمنية **deadlines** هي عملياً لحظات على محور الزمن. والسؤال الذي يطرح نفسه هنا هو: "**من أين تأتي الحدود الزمنية وما الذي يحدد قيمها؟**" يتعلق الجواب بالظواهر الفيزيائية التي يعتمد عليها نظام التحكم المدروس. فعلى سبيل المثال، يجب تحديث الصورة **30** مرة في الثانية في نظم العرض (كالشاشات وأجهزة الإسقاط) للحصول على صورة متحركة ذات جودة مقبولة لأن العين الإنسانية في هذه الحالة لا تحس بانقطاع الحركة بسبب دوام الانطباع الشبكي. إذا لم يحقق جهاز العرض هذه السرعة، تصبح حركة الصورة مقطعة. الظاهرة الفيزيائية التي يعتمد عليها النظام هنا هي العين الإنسانية. وفي نظم الملاحة المختلفة، يجب حساب تسارع الجسم المدروس (طائرة، سيارة، صاروخ، ...) بمعدل يعتمد على السرعة القصوى للجسم. لهذا فإن معدل حساب تسارع السيارة أقل من معدل حساب سرعة الصاروخ لأن السرعة القصوى للسيارة أقل بكثير من السرعة القصوى للصاروخ.

تعمل نظم الزمن الحقيقي في بعض الحالات تحت قيود زمنية فرضتها متطلبات تخمينية أو غير مهمة أصلاً، مما يجعل هذه القيود أقسى بكثير من الحاجة. لذلك فإنه من الضروري قبل تصميم أي نظام زمن حقيقي دراسة القيود الزمنية التي تفرضها متطلبات النظام المهمة ومحاولة تخفيفها

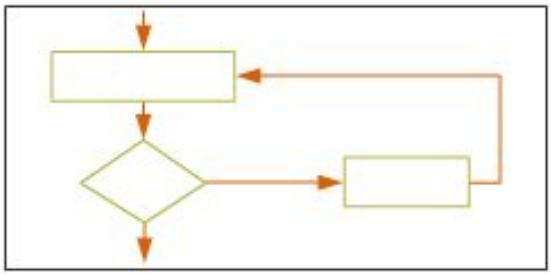
---

[2] سيدور معظم نقاشنا في هذا الكتاب حول نظم الزمن الحقيقي الصعبة hard real-time systems. لذلك، سنستعمل المصطلح "**نظام زمن حقيقي**" ليدل على نظام زمن حقيقي **صعب ومضمن** ما لم نحدد غير ذلك.

قدر الإمكان بما يناسب البيئة الفيزيائية التي يتحكم النظام بها.

الأحداث events والحتمية determinism

إن الأسلوب الطبيعي لتنفيذ التعليمات في أي نظام برمجي **software** هو التنفيذ التسلسلي. يحتوي المعالج الذي ينفذ هذا البرنامج سجلاً خاصاً يسمى عادةً "**سجل عداد البرنامج**" **Program Counter Register** يحتوي بدوره على عنوان التعليمة التي ينفذها المعالج حالياً. عندما يكون تنفيذ البرنامج تسلسلياً، تجري زيادة قيمة هذا السجل بمقدار محدد للحصول على عنوان التعليمة التالية بعد الانتهاء من تنفيذ التعليمة الحالية. لكن هناك بعض الحالات التي تغير هذا التنفيذ التسلسلي وتجعل المعالج يقفز لمكان آخر في البرنامج لتنفيذ التعليمات الموجودة فيه. والمثال التقليدي على مثل هذه الحالات هو تنفيذ تعليمة **if-then-goto** المبينة على شكل معين في مخطط التدفق التالي:



من الأمثلة الأخرى على الحالات التي تغير من التدفق التسلسلي للتنفيذ: استدعاء التوابع في اللغات الإجرائية مثل **C** وباسكال وإنشاء الأغراض **objects** واستدعاء طرائقها **methods** في اللغات الغرضية التوجه **Object Oriented Languages** مثل **C++** و **Java**.

الحدث event: هو أي شيءٍ يحدث ويؤدي لتغير عداد البرنامج بأسلوبٍ غير تسلسلي (وهو ما نعتبره تغيراً في تدفق البرنامج).

عادةً ما تتكون نظم الزمن الحقيقي من عددٍ من الإجراءات **Tasks or Jobs** تعمل على التفرع وكل منها يؤدي عملاً محدداً. ويمكن أن تكون هذه الإجراءات دوريةً **periodic** أو غير دورية **aperiodic** (تحدث استجابةً لحدثٍ ما). فعلى سبيل المثال، الإجراء الذي يقوم بالحفاظ على سرعة الطائرة في الجو هو إجراء دوري يقوم دورياً بقراءة سرعة الطائرة الحالية ويحسب الفرق بينها وبين السرعة المطلوب الحفاظ عليها ويرسل على أساسها أوامر التحكم لمحركات الطائرة. أما في نظام مراقبة الحريق فيؤدي ارتفاع درجة الحرارة لإقلاع إجراء إطفاء الحريق، فهو إذاً إجراء غير دوري.

لحظة الإطلاق Release time لإجراء: هي اللحظة التي يصبح فيها هذا الإجراء جاهزاً للتنفيذ (بشكل دوري أو غير دوري).

وغالباً ما تتعلق لحظة الإطلاق بمقاطعةٍ **interrupt** ما. إما أن تكون هذه المقاطعة ناتجةً عن مؤقت دوري **timer** كما هو الحال في الإجراءات الدورية، أو أحداث محددة كارتفاع درجة الحرارة مثلاً في نظام مراقبة الحريق.

هناك نوعان من الأحداث:

- **الأحداث المتزامنة synchronous:**

تحدث في لحظات يمكن التنبؤ بها في تدفق تنفيذ البرنامج، مثل تعليمة القفز الشرطي (المعين) المبينة في مخطط التدفق السابق. ورغم أن القفز المشروط لا يتحقق دائماً، إلا أنه يمكن التنبؤ به.

- **الأحداث الغير متزامنة asynchronous:**

تحدث في لحظات لا يمكن التنبؤ بها طوال فترة تنفيذ البرنامج وتسببها مصادر من خارج النظام.

تجدر الإشارة إلى أننا نعتبر المؤقت الذي يولد نبضاتٍ منتظمة (كل **5** ميلي ثانية على سبيل المثال) حدثاً غير متزامن. فعلى الرغم من أنه يمثّل حدثاً دورياً (بافتراض أن المؤقت مثالي ودقيق تماماً وهو أمر مستحيل عملياً)، فإن اللحظات التي سيولد فيها مقاطعةً لتدفق تنفيذ البرنامج تخضع لعوامل عديدةٍ لا يمكن التنبؤ بها، مثل اللحظة التي بدأ فيها المؤقت بالعمل نسبةً لبداية تنفيذ البرنامج وزمن تأخير انتشار الإشارة في النظام الحاسوبي نفسه وغيرها من العوامل. لذلك لا يمكن التنبؤ بلحظات المقاطعة تلك.

ذكرنا سابقاً أن الأحداث التي لا تحدث بدور منتظم تسمى أحداثاً غير دورية **aperiodic**. تسمى الأحداث الغير دورية والتي قلما تحدث بالأحداث المتباعدة **sporadic**.

يبين الجدول التالي ملخصاً لتصنيف الأحداث وبعض الأمثلة عليها:

	دورية periodic	غير دورية aperiodic	متباعدة sporadic
متزامنة synchronous	تعليمات برنامج في حلقة، إجراءات مجدولة باستعمال مؤقت داخلي.	تعليمة قفز تقليدية، مجمع النفايات في اللغات الحديثة garbage collection.	تعليمة قفز لمعالجة الاستثناءات (أخطاء زمن التنفيذ) exception or traps.
غير متزامنة asynchronous	مقاطعات مولدة من مؤقت خارجي.	مقاطعات منتظمة ولكن ليس لها دور ثابت.	استثناءات مولدة خارجياً، أحداثٌ عشوائية.

من المهم في أي نظام، وخصوصاً في النظم المضمنة، أن يبقى النظام المتحكم به تحت سيطرة نظام التحكم باستمرار. يمكن أن تحدث بعض الحالات الخاصة التي يخرج فيها النظام المتحكم به عن السيطرة. يجب على برنامج التحكم أن يتوقع هذه الحالات ويتعامل معها ليعيد السيطرة على النظام.

في أي نظام زمن حقيقي، يحافظ برنامج التحكم على السيطرة إذا كان من الممكن التنبؤ بالحالة التالية للنظام إذا أُعطِينا حالته الحالية وقيم مداخله. أي أن هدفنا الأساسي للحصول على نظام تحكمٍ صحيح هو توقع كل الظروف الممكنة وجعل النظام مستعداً للتصرف بشكلٍ صحيح حين تحدث.

النظام الحتمي deterministic: نقول عن نظام أنه حتمي إذا كان من الممكن معرفة حالته وقيم مخارجه التالية مهما كانت حالته وقيم مداخله الحالية.

حتمية الأحداث event determinism: تعني أنه يمكن معرفة حالة وقيم مخارج النظام التالية من أجل كل قيم ممكنة للمداخل التي تقدح trigger الأحداث. لاحظ أن أي نظامٍ حتمي هو حتمي الأحداث، لكن العكس ليس صحيحاً بالضرورة، وإن كان من الصعب أن يكون النظام حتمي الأحداث ولكنه غير حتمي.

الحتمية الزمانية temporal determinism: يتصف النظام بالحتمية الزمانية إذا كان حتمياً وكان زمن الإستجابة لكلّ حالةٍ ممكنة من حالاته معروفاً.

من المفضل طبعاً أن يتصف أي نظام تحكمٍ بالحتمية لأن ذلك يضمن استجابته في أي وقت

ومهما كانت الظروف. وإذا اتصف بالحتمية الزمانية، يضمن ذلك معرفة متى سيستجيب النظام، وهذا أقرب ما يكون لنظم الزمن الحقيقي.

انشغالية المعالج CPU utilization

سنعرف الآن مصطلحاً هاماً يستعمل لقياس أداء نظم الزمن الحقيقي. يقوم المعالج باستمرار بجلب التعليمات من الذاكرة وفك ترميزها وتنفيذها طالما بقي مغذّى بالتيار الكهربائي. إما أن تكون هذه التعليمات هي من تعليمات برنامج التحكم المفيدة، أو تعليمات أقل أهمية ليس لها علاقة بالحدود الزمنية للنظام (مثل تعليمات التفاعل مع المستخدم على سبيل المثال)، أو تعليمات **NOP** (وهي تعليمة آلة لا يؤدي تنفيذها لأي نتائج وهي اختصار لـ **No-Operation**). يدل قياس الزمن الذي يقضيه المعالج في وضع الخمول **Idle** (وهو عندما ينفذ المعالج تعليمات **NOP** فقط لأنه لا يوجد ما يقوم به) على مدى المعالجة في الزمن الحقيقي التي يتطلبها النظام.

انشغالية المعالج (U) **CPU utilization** أو **معامل تحميله الزمني time-loading factor:** هي النسبة الزمنية التي لا يكون المعالج فيها في وضع الخمول.

- نقول عن نظامٍ أنّه محملٌ تحميلاً زائداً إذا كانت قيمة **U** أكبر من **100%**.
- من غير المفضل أن تكون انشغالية النظام عالية، لأن أي تغيير أو إضافة لمهام جديدة يمكن أن تؤدي للتحميل الزائد.

كذلك فإن النظم ذات الانشغالية المنخفضة ليست بالضرورة جيدة، لأنها تدل على سوء التصميم الهندسي للنظام وأنه كان من الممكن تخفيض كلفة النظام باستعمال تجهيزاتٍ مادية أقلّ أداء وكلفة.

عادةً ما تكون الانشغالية في النظم الحديثة حوالي **50%**، بينما يمكن لبعض النظم التي لا يتوقع نموها أن تكون انشغاليتها بحدود **80%**.

على كل حال، تُعتبر القيمة الأمثلية للانشغالية في نظم الزمن الحقيقي ذات المهام الدورية والمستقلة هي **70%**.

ويبين الجدول التالي بعض القيم الممكنة للانشغالية والحالات النموذجية الموافقة لها:

الانشغالية (%)	المنطقة التي يعمل فيها النظام	التطبيقات النموذجية
0-25	إمكانيات معالجة زائدة جداً – المعالج أقوى بكثير من الحاجة	متعددة
26-50	آمنة جداً	متعددة
51-68	آمنة	متعددة
69	الحد النظري	النظم المضمنة
70-82	قريبة من الخطر	النظم المضمنة
83-99	خطيرة	النظم المضمنة
100+	تحميل زائد	النظم المجهدة

يمكن حساب U بجمع معاملات انشغاليات الإجراءات الدورية وغير الدورية المشاركة. وبفرض أن النظام يحتوي على $n \geq 1$ إجراء دوري دور كل منها يساوي P_i (وبالتالي يكون معدل تنفيذها $F_i = \frac{1}{P_i}$). فإذا علمنا أن زمن التنفيذ الاعظمي التقديري (في أسوأ الأحوال) للإجراء i هو e_i، فإن معامل الانشغالية u_i يحسب كما يلي:

$$u_i = \frac{e_i}{P_i} \qquad \textbf{(1.1)}$$

وبذلك تكون الانشغالية الكلية U للنظام مساويةً لـ :

$$U = \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} \frac{e_i}{P_i} \qquad \textbf{(1.2)}$$

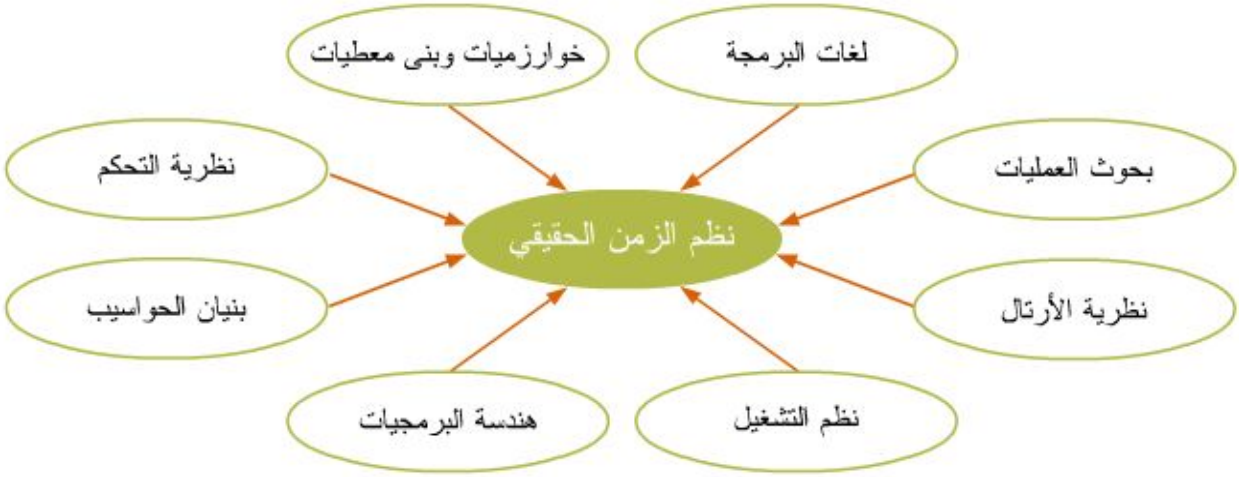
عادةً ما يكون الحد الزمني **deadline** للإجراء الدوري رقم i (ونرمزه d_i) محدوداً ببداية دوره التالي. القيمة المنطقية العظمى لـ d_i هي e_i. يمكن أن تكون عملية تحديد قيمة e_i قبل أو بعد تنفيذ البرنامج صعبةً جداً أو مستحيلةً في بعض الأحيان، عندها نقدر قيمتها تقديراً. في حال الإجراءات الغير دورية **aperiodic** والمتباعدة **sporadic**، نحسب u_i بدلالة الحالة الأسوأ لزمن تنفيذها والزمن الفاصل بين لحظات حدوثها. تختلف انشغالية المعالج عن أدائه **throughput**، الذي يعرف بأنه عدد تعليمات الآلة التي ينفذها المعالج في الثانية.

عادةً ما يستعمل هذا القياس لمقارنة أداء المعالجات المختلفة عند تنفيذها لتطبيق محدد.

عوامل تتعلق بتصميم نظم الزمن الحقيقي

تعتبر نظم الزمن الحقيقي فرعاً معقداً من هندسة النظم الحاسوبية وتتأثر بكل من نظرية التحكم وهندسة البرمجيات وبحوث العمليات (بسبب الحاجة لجدولة الإجراءات **scheduling**).

يبين الشكل التالي بعض فروع هندسة الحاسوب والإلكترونيات التي تؤثر بتحليل وتصميم نظم الزمن الحقيقي:



يجب الانتباه إلى عدة مشاكل يجب حلها عند تصميم وتحقيق نظم الزمن الحقيقي وتتضمن:

- اختيار العتاد المادي والبرمجيات وإيجاد الحل الأمثل بالكلفة الأقل.
- تحديد مواصفات النظام وتصميمه واختيار التمثيل الأفضل لتصرفه الزمني.
- فهم الفروق بين لغات البرمجة المختلفة ومدى ملاءمتها لنظم الزمن الحقيقي.
- تصميم النظام بحيث تكون وثوقيته وتحمله للأخطاء أكبر ما يمكن.
- اختبار النظام بدقة.
- الاستفادة من تقنية النظم المفتوحة والمتوافقة قدر الإمكان. النظام المفتوح هو مجموعة مستقلة وقابلة للتوسع من التطبيقات التي تتعاون لتعمل كنظام متكامل. على سبيل المثال، يستعمل حالياً عدد من نسخ نظام التشغيل المفتوح لينوكس **Linux** في تطبيقات الزمن الحقيقي. تُقاس التوافقية بمدى تحقيق النظام لمقاييس النظم المفتوحة مثل مقياس كوربا **CORBA** لنظم الزمن الحقيقي.
- قياس وتقدير أزمان الاستجابة للإجراءات ومحاولة تصغيرها قدر الإمكان، وإجراء تحليل لإمكانية جدولة الإجراءات، أي إمكانية إيجاد ترتيب ما لجدولة الإجراءات بحيث تحترم جميعها حدودها الزمنية. تدخل هذه الدراسة في نطاق نظرية الجدولة، وهي جزء من بحوث العمليات.

يمكن بالطبع استعمال التقانات الهندسية لنظم الزمن الحقيقي الصعبة **hard real-time** لهندسة الأنواع الأخرى من نظم الزمن الحقيقي للحصول على نظم أفضل وأكثر وثوقية ومتانة. وهذه

أحد أسباب أهمية دراسة نظم الزمن الحقيقي.

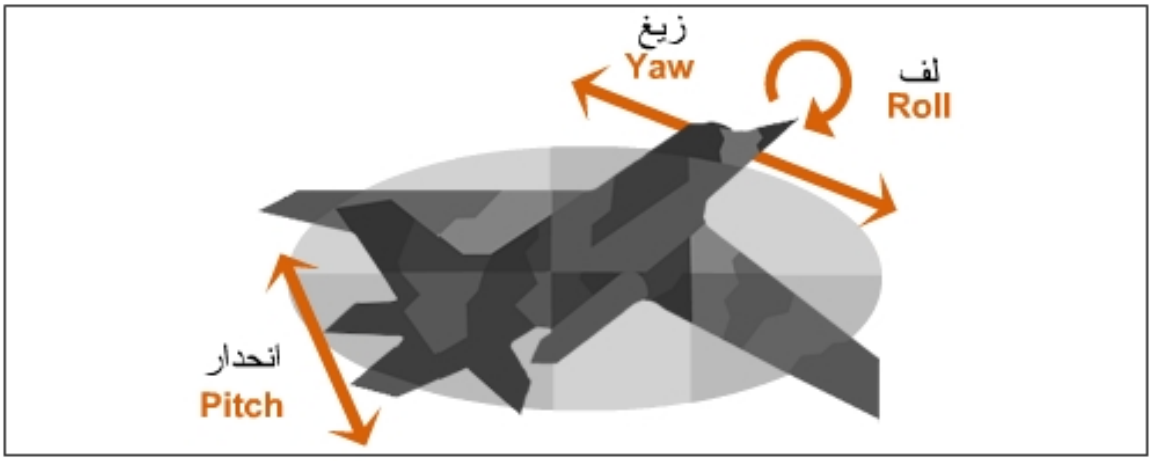
أمثلة على نظم الزمن الحقيقي

إن نظم الزمن الحقيقي المضمنة متنوعة جداً وتوجد حتى في الألعاب والأجهزة المنزلية. ويبين الجدول التالي بعض العينات من نظم الزمن الحقيقي ومجالات تطبيقاتها:

المجال	التطبيقات
الطيران	الملاحة شاشات الإظهار
تعدد الوسائط	الألعاب نظم المحاكاة
الطب	جراحة الروبوتات الجراحة البعيدة التصوير الطبي
النظم الصناعية	خطوط التجميع المؤتمتة التفتيش الآلي
النظم المدنية	المصاعد السيارات

سنقدم فيما يلي تفصيلاً لبعض الأمثلة المذكورة في هذا الجدول. ليس المقصود هنا إعطاء توصيف صوري لهذه النظم، بل توصيفها بطريقة تظهر أهمية نظم الزمن الحقيقي في حياتنا العملية. سنناقش لاحقاً طريقة توصيف هذه النظم بطريقة دقيقة وصورية.

لنعتبر نظام القياس العطالي في الطائرة. يبين الشكل التالي الحركات الممكنة التي يمكن أن تقوم بها الطائرة والتي يجب على النظام قياسها:



حسب مواصفات النظام، ترسِل تجهيزات خاصة نبضات مقياس التسارع على المحاور **x** و **y** و **z** إلى برنامج التحكم كل **10** ميلي ثانية. يحسب البرنامج تسارع الطائرة على المحاور الثلاثة، بالإضافة إلى قيم اللاف والانحدار والزيغ الحالية للطائرة. يصل كذلك للبرنامج قيمة درجة الحرارة كل ثانية. يجب عليه حساب شعاع السرعة اعتماداً على هذه المعطيات كل **40** ميلي ثانية وإظهارها للطيار.

لنناقش نظام مراقبة لمفاعل نووي عليه التعامل مع ثلاثة أحداث يدلّ عليها بمقاطعات. يحدث الحدث الأول عندما تكتشف نقاط المراقبة الأمنية أي اختراق أمني للمفاعل. يجب على النظام في هذه الحالة الاستجابة خلال ثانية واحدة لإطلاق أجهزة الإنذار. يدل الحدث الثاني والأهم على حدوث ارتفاع كبير لدرجة الحرارة في المفاعل، ويجب الاستجابة له خلال ميلي ثانية واحدة فقط. أخيراً، يجب تحديث شاشة الإظهار **30** مرة في الثانية.

في نظام حجز التذاكر لشركة طيران. قررت إدارة الشركة أنه يجب ألا يتجاوز زمن الانتظار الكلي لأي عملية حجز **15** ثانية حتى لا يؤدي ذلك إلى عدم رضا الزبائن. يجب كذلك ألا يسمح النظام بالحجز الزائد عن عدد المقاعد المتوفرة. يمكن لعدة وكلاء أن يحاولوا الوصول إلى قاعدة المعطيات في نفس الوقت لحجز نفس المقعد. لذلك يجب تزويد النظام بإمكانيات الاتصال والقفل اللازمة لذلك.

والمثال الأخير هو نظام تحكم بإشارات المرور على تقاطع طرق ترد إليه السيارات من أربعة اتجاهات (شمال، جنوب، شرق وغرب). يتحكم النظام بإشارات السيارات والمشاة ويتلقى المعلومات من العالم الخارجي عن طريق حساسات تحت الأرض لقياس معدل تدفق السيارات

في كل اتجاه، بالإضافة إلى كاميرات المراقبة والأزرار التي يضغطها المشاة عند رغبتهم بقطع الشارع. يجب على إشارات المرور أن تعمل بتزامن تام والاستجابة كذلك لأحداث غير متزامنة مثل الضغط على زر المشاة، وإلا حدثت حوادث مميتة.

بعض المعتقدات الخاطئة المتعلقة بنظم الزمن الحقيقي

من المهم أن نذكر بعض المعتقدات الشائعة الخاطئة لفهم طبيعة نظم الزمن الحقيقي فهماً كاملاً. ونلخص فيما يلي هذه المعتقدات:

- **"نظم الزمن الحقيقي" هي من مرادفات "النظم السريعة":** يأتي هذا الاعتقاد الخاطئ من حقيقة أن العديد من نظم الزمن الحقيقي تتعامل حقاً مع حدود زمنية من رتبة الميلي ثانية، كما هو حال نظم الملاحة الجوية. لكن هناك كذلك نظم زمنٍ حقيقي أخرى بطيئة مثل نظام حجز تذاكر الطيران. المهم في نظم الزمن الحقيقي هو النجاح في تحقيق الحدود الزمنية وليس السرعة.
- **تحل خوارزمية الجدولة RM (Rate Monotonic) مشكلة الزمن الحقيقي:** تعتبر هذه الخوارزمية من أكثر خوارزميات جدولة الإجراءات الدورية شهرةً في أدبيات نظم الزمن الحقيقي منذ عام **1970**. وهي بسيطة وتعطي عادةً نتائج جيدة، لكنها ليست دائماً الحلّ السحري لنظم الزمن الحقيقي. سندرس هذه الخوارزمية بالتفصيل في الفصول اللاحقة.
- **هناك طرق عامة ومقبولة دائماً لتصميم ووضع مواصفات نظم الزمن الحقيقي:** لا توجد للأسف مثل هذه الطرق وذلك لصعوبة إيجاد حلولٍ عامة لكلّ الحالات. يجب دراسة وتصميم كلّ نظامٍ على حدةً حسب خصوصيته.
- **لا حاجة لبناء نظم تشغيل الزمن الحقيقي من الصفر وذلك لتوفر الكثير من الحلول التجارية:** صحيح أن هناك العديد من نظم تشغيل الزمن الحقيقي الشائعة ذات الكلفة المقبولة، لكنها ليس مناسبة لكل تطبيقات الزمن الحقيقي. نحتاج أحياناً لكتابة نظام تشغيلٍ مناسبٍ لتطبيقٍ محدد من الصفر، أو تعديل نظام تشغيل جاهز ليتناسب مع خصوصية التطبيق. كذلك فإن اختيار نظام التشغيل المناسب من بين النظم المتوفرة ليس بالأمر السهل.
- **تتعلق معظم دراسة نظم الزمن الحقيقي بدراسة نظرية الجدولة:** غالباً ما تكون النتائج التي يحصل عليها الباحثون في نظرية الجدولة غير عملية وليس لها قيمة كبيرة هندسياً. لذلك سنناقش في هذا المقرر الخوارزميات القابلة للتطبيق عملياً فقط.

2. لمحة تاريخية مختصرة

يرتبط تطور نظم الزمن الحقيقي بشكلٍ أو بآخر بتطور الحاسوب.

إن العديد من نظم الزمن الحقيقي الحديثة مثل نظم التحكم بمحطات توليد الطاقة النووية ونظم أسلحة الطائرات والصواريخ هي نظم معقدة جداً، لكن يرتكز الكثير منها على مفاهيم كلاسيكية طورت ما بين الأربعينيات والستينيات من القرن العشرين.

التطورات النظرية

إن معظم نظريات نظم الزمن الحقيقي مشتقة من نظريات فروع العلوم الأخرى التي تعتمد عليها والتي ذكرناها سابقاً، وتأثرت بشكل خاص بالتطورات النظرية لبحوث العمليات التي حصلت في نهاية الأربعينيات ونظرية الأرتال **Queuing Theory** التي تطورت في نهاية الخمسينيات.

تبلورت النتائج النظرية المتعلقة بوثوقية وتنبؤية **predictability** نظم الزمن الحقيقي في الثمانينيات والتسعينيات من القرن العشرين. وجرت كذلك في هذه الفترة دراسة المشاكل المتعلقة بنظم الزمن الحقيقي متعددة المعالجات **multiprocessing**.

بعض نظم الزمن الحقيقي المبكرة

يمكن اعتبار أن بداية ظهور مفهوم نظام الزمن الحقيقي بشكلٍ واضح كان في النظامين **Whirlwind** (وهو نظام محاكاة طيران طورته شركة **IBM** لمصلحة البحرية الأمريكية **US Navy** في عام **1947**) و **SAGE** (وهو نظام محاكاة للدفاع الجوي طور لمصلحة القوى الجوية الأمريكية **US Air Force** في بدايات الخمسينيات). يحقّق هذان النظامان المعايير التي تجعلهما يصنفان كنظم زمنٍ حقيقي فعلية.

من الأمثلة الأخرى على نظم الزمن الحقيقي المبكرة نجد النظام **SABRE**، وهو نظام حجز للخطوط الجوية طورته الخطوط الجوية الأمريكية عام **1959**.

برمجيات الزمن الحقيقي المبكرة

كُتِبت برمجيات الزمن الحقيقي الأولية بلغة الآلة أو لغة التجميع **Assembly**، ثم استُعمِلت لاحقاً اللغات ذات المستوى الأعلى لهذا الغرض مثل **Fortran** و **CMS-2**.

تبنّت إدارة الدفاع الأمريكية **Department of Defense** في السبعينيات من القرن العشرين فكرة تطوير لغة برمجةٍ موحدة يمكن استعمالها لبرمجة جميع الخدمات الممكنة، وتؤمن البنى

البرمجية عالية المستوى الضرورية لبرمجة نظم الزمن الحقيقي. ظهرت لغة **ADA** نتيجة لذلك عام **1983** بعد عدة عمليات انتقاء وتحسين، وكان من المتوقع أن تكون هي اللغة المثلى لجميع التطبيقات. في العام **1995**، صدرت نسخة محسنة من **ADA** سميت **ADA-95** تحلّ بعض المشاكل التي اكتشِفت في النسخة الأصلية.

لكن يستعمل عدد قليل فقط من المبرمجين لغة **ADA-95** اليوم، بينما يستعمل الأغلبية لغة **C** أو **C++** أو حتى **Fortran** ولغة التجميع. يعود السبب على الأغلب لتعقيد لغة **ADA-95** وعدم توفر مترجماتٍ لها على نظم التشغيل الشائعة. وفي الآونة الأخيرة، ظهر توجه لاستعمال لغات البرمجة غرضية التوجه (بالأخص جافا **JAVA**) في نظم الزمن الحقيقي المضمنة. سنناقش لغات البرمجة لنظم الزمن الحقيقي لاحقاً في فصلٍ مستقل.

نظم تشغيل الزمن الحقيقي التجارية

صممت نظم تشغيل الزمن الحقيقي التجارية الأولية للعمل على الحواسب العملاقة **mainframes**، مثل نظام **Basic Executive** الذي طورته شركة **IBM** في العام **1962** والذي يحقق العديد من خوارزميات الجدولة في الزمن الحقيقي. وفي العام **1963**، صدر النظام **Basic Executive II** الذي سمح بأن تقيم برامج المستخدم والنظام على القرص.

ظهر بعد ذلك في السبعينيات حواسب أصغر **minicomputers** وأقلّ سعراً وانتشرت في الكثير من الأوساط الهندسية. ونتيجةً لذلك ظهر عدد كبير من نظم تشغيل الزمن الحقيقي الهامة التي طورتها نفس الشركات المنتجة لهذه الحواسيب، مثل نظام **Real-time Multitasking (RSX) Executives** الذي طورته شركة **Digital Equipment Corporation (DEC)** لحواسبها من نوع **PDP-11**. ومن الأمثلة الأخرى نظام **RTE** الذي طورته شركة **Hewlett-Packard** لسلسة حواسيبها من نوع **HP 2000**.

في أواخر السبعينيات وأوائل الثمانينيات، ظهرت نظم التشغيل المخصصة للمعالجات الصغرية **micro processors** مثل الأنظمة **RMX-80** و **MROS 68K** و **VRTX** وغيرها. خلال العشرين سنة الماضية، ظهر الكثير من نظم تشغيل الزمن الحقيقي واختفى العديد منها أيضاً.

مقرر: نظم الزمن الحقيقي

الفصل الثاني: نظم تشغيل الزمن الحقيقي

فهرس الفصل:

1. نواة نظام تشغيل الزمن الحقيقي

الإجراء process (ويسمى أيضاً المهمة task): هو برنامج قيد التنفيذ، وهو الوحدة المنطقية التي يجدولها[1] نظام التشغيل.

ويمثل كل إجراء في نظام التشغيل ببنية معطيات (تسجيلة **record**) تحتوي على الأقل على حالة الإجراء الحالية (جاهز **ready**، مجمد **blocked**، في حالة تنفيذ **executing**، ...) ورقم تعريف للإجراء **identity number**، وبعض الواصفات الأخرى (زمن التنفيذ الكلي، ...) ومجموعة المصادر (ملفات، أجهزة، ...) التي يحجزها هذا الإجراء.

النيسب thread: هو إجراء خفيف **lightweight** يتشارك بالمصادر مع بعض النياسب الأخرى.

ويتألف كل إجراء من عدة نياسب (واحد على الأقل) تشاركه مصادره التي يحجزها. يصعب عادةً مشاركة المصادر بين عدة إجراءات، لكن المشاركة سهلة بين النياسب المكونة لنفس الإجراء.

يجب على نظام تشغيل الزمن الحقيقي أن يؤمن ثلاثة خدمات للإجراءات: الجدولة **scheduling** والتوزيع **dispatching** والاتصال والتزامن بين الإجراءات **intercommunication and synchronization**. ونواة نظام التشغيل **kernel** هي الجزء الأصغر من نظام التشغيل الذي يؤمن هذه المهام الثلاث.

كما ذكرنا سابقاً، يحدد المجدول **scheduler** الإجراء الذي سينفذ في نظام تشغيل متعدد المهام **multitasking**، بينما يقوم الموزع **dispatcher** بالعمليات التنظيمية اللازمة لتنفيذ هذا الإجراء. وأخيراً، يضمن الجزء الثالث أن الإجراءات تتعاون وتتواصل فيما بينها. ويبين الشكل التالي طبقات نظام التشغيل المختلفة والمهام التي تقوم بها:

---

[1] **الجدولة scheduling:** هي عملية انتقاء الإجراء الذي سيقوم المعالج بتنفيذه في لحظة محددة من بين كل الإجراءات الجاهزة للتنفيذ.

المستخدم	
برنامج **shell**	نظام التشغيل
دعم للأقراص والملفات	التنفيذي **Executive**
الاتصال والتزامن بين الإجراءات	النواة
جدولة الإجراءات	النواة الصغرية **micro-kernel**
إدارة النياسب	النانو – نواة **nano-kernel**
العتاد المادي	

تؤمن النانو-نواة مهمة إدارة النياسب فقط، بينما تهتم النواة الصغرية بجدولة الإجراءات. تؤمن النواة عمليات التزامن والاتصال بين الإجراءات باستعمال العلامات **semaphore** وصناديق البريد **mailboxes** وغيرها من الطرق الأخرى. الجزء "التنفيذي" **Executive** من نظام تشغيل الزمن الحقيقي هو نواة تتضمن عمليات إدارة كتل الذاكرة وأجهزة الدخل/خرج وغيرها من العمليات المعقدة. معظم أنظمة تشغيل الزمن الحقيقي التجارية هي فقط هذا الجزء "التنفيذي" ولا تتضمن الطبقات الأدنى. أخيراً، تؤمن طبقة نظام التشغيل الأعلى واجهات التخاطب مع المستخدم والأمن وإدارة الملفات وغيرها.

مهما كان بنيان نظام التشغيل المستعمل، يبقى الهدف الأساسي منه هو تلبية متطلبات الزمن الحقيقي وتأمين بيئة عمل متعددة المهام متينة ومرنة.

- أشباه النوى **Pseudo-Kernels**.
- النظم المقادة بالمقاطعة **Interrupt-Driven Systems**.
- نظم الأولوية الشُفعية **Preemptive-Priority**.
- النظم الهجينة **Hybrid Systems**.
- نظم الواجهة الأمامية/الواجهة الخلفية **Foreground/Background**.
- نظم تشغيل الزمن الحقيقي كاملة المواصفات.

أشباه النوى Pseudo-Kernels

يمكن الحصول على إمكانيات الزمن الحقيقي بحلولٍ بسيطة دون الحاجة لدعمٍ كبيرٍ من نظام التشغيل. وهذه الحلول مفضلة إذا وجدنا أنها تكفي في تطبيق ما، لأن نظام التحكم الناتج أبسط وأسهل تحليلاً.

سنستعرض فيما يلي أهم هذه الأساليب:

- حلقة الاستقصاء **Polling Loop**.
- حلقة الاستقصاء المتزامنة **Synchronized Polling Loop**.
- التنفيذي الدوري **Cyclic Executive**.
- البرمجة المقادة بالحالة **State-Driven Coding**.
- الروتينات التعاونية **Co-Routines**.

حلقة الاستقصاء Polling Loop

عادةً ما تُستَعمل حلقة الاستقصاء للاستجابة السريعة لجهاز واحد.

وهي تتألف من حلقة غير منتهية تحوي تعليمة اختبار متكرر لعلام **flag** يدل على حدوث أو عدم حدوث حدث ما. تسمى عملية الفحص المتكررة هذه "استقصاء" **polling**.

```
for ( ; ; ) {                          /* do forever */
        if (packet_here) {             /* check flag */
                process_data();        /* process
data */
                packet_here = 0;       /* reset flag */
        }
```

على سبيل المثال، نفترض أننا بحاجة لنظام برمجي لمعالجة حزم **packets** من المعطيات بمعدل لا يزيد عن حزمة واحدة في الميلي ثانية. نفترض أنه لدينا علام اسمه **packet_here** تصبح قيمته **1** عندما يستلم كرت الشبكة حزمة من المعلومات.

يبين البرنامج البسيط التالي حلقة استقصاء مكتوبة بلغة **C** يتعامل مع حزم المعطيات الواردة:

تناسب هذه الطريقة حالة نظام مكون من معالج وحيد مخصص للتعامل مع دخل/خرج سريع.

عادةً ما تُستعمل هذه الطريقة كإجراء خلفي **background task** في النظم المقادة بالمقاطعات، أو كإجراء دوري. تدور حلقة الاستقصاء في هذه الحالة عدداً قليلاً من المرات كي لا تستأثر بالمعالج ولتسمح لباقي الإجراءات بالتنفيذ، وتقوم باقي الإجراءات بتنفيذ المعالجة التي لا علاقة لها بالحدث.

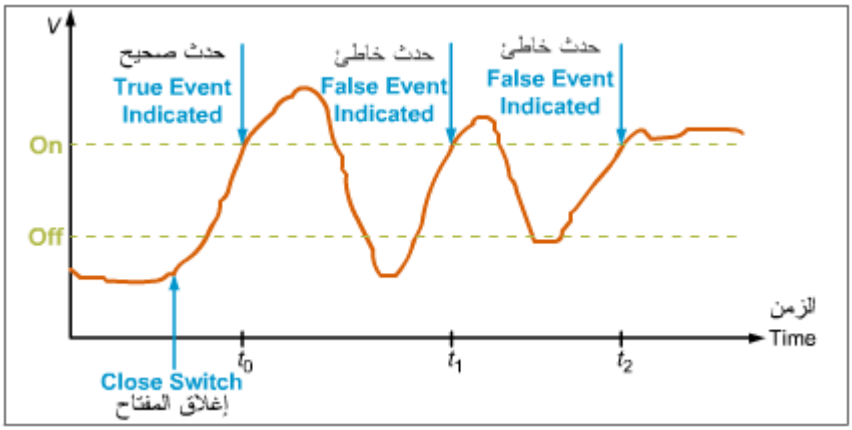
حلقة الاستقصاء المتزامنة Synchronized Polling Loop

هذه الطريقة مشابهة للطريقة السابقة، لكنها تضيف تأخيراً زمنياً بين لحظة الاستجابة للحدث ولحظة تصفير **reset** العلام (أي قبل السطر **"packet_here=0;"** في المثال السابق).

يستفاد من هذه الخوارزمية للاستجابة للأحداث التي تتصف بما يسمى "بارتداد المفتاح" **switch bounce**. تحدث هذه الظاهرة عندما يكون مسبب الحدث مفتاحاً ميكانيكياً أو كهربائياً، لأنه من المستحيل صنع مفتاح يغير حالته لحظياً دون اهتزاز لتماساته.

يبين الشكل مخططاً زمنياً لمفتاحٍ لحظة إغلاق تماساته.

يظهر جلياً من المخطط أن اهتزاز تماسات المفتاح سبب عدة أحداث متتالية، الأول منها حـدث في اللحظة t_0 وهو صحيح والأحداث التالية التي حدثت في اللحظات t_1 و t_2 هي خاطئة. عند إضافة تأخيرٍ زمني كافٍ (أكبر من الفرق الزمني بين t_0 و t_2) قبل تصفير علام الحدث، لـن تسبب الأحداث الخاطئة استجابات من النظام لأنه يكون عندها في حالة انتظار انقـضاء فتـرة التأخير الزمني.



على سبيل المثال، إذا فرضنا أن ارتداد المفتاح يزول بعد **20** ميلي ثانية، يمثّل البرنامج التالي المكتوب بلغة **C** حلقة استقصاء متزامنة مناسبة له. تعليمة **pause(20)** هي استدعاء لخدمة من نظام التشغيل تجمد تنفيذ البرنامج لمدة **20** ميلي ثانية.

```
for ( ; ; ) {                          /* do forever */
        if (flag) {                    /* check flag */
                process_data();        /* process
data */
                pause(20);             /* wait 20 ms
*/
                flag = 0;              /* reset flag */
```

تتميز حلقات الاستقصاء بأنها مناسبة جداً للاستجابة لأقنية المعطيات السريعة، وخاصةً عندما تحدث الأحداث بلحظات غير منتظمة ويكون المعالج مخصصاً لمعالجتها فقط. لكنها غير كافية عندما يكون النظام معقداً، وهي تضيع الكثير من وقت المعالج وخصوصاً عندما تكون الأحداث قليلة الحدوث.

التنفيذي الدوري Cyclic Executive

نظام "التنفيذي الدوري" هو نظام غير مقاد بالمقاطعات (أي لا يعتمد على المقاطعات في عمله) ويعطي انطباعاً بالتنفيذ المتوازي لعدة إجراءات قصيرة على معالج سريع.

ويتألف النظام من حلقة غير منتهية تستدعي الإجراءات على التتالي كما هو مبين في المثال التالي:

```
for ( ; ; ) {                    /* do
forever */
        Process_1();
        Process_2();
        ...
        Process_n();
        }
```

نلاحظ في هذا المثال أن زمن المعالج موزع بالتساوي على جميع الإجراءات لأنه يدور عليهم بأسلوب "الشريط الدوار" **round-robin**.

يمكن بسهولة تعديل الحلقة السابقة لإعطاء إجراءات محددة أزمان تنفيذ أكبر (وبالتالي أولوية وسرعة استجابة أكبر) كما هو مبين في المثال التالي الذي يعطي الإجراء **Process_3** زمن تنفيذ يساوي ضعف زمن تنفيذ الإجراءات الأخرى:

```
for ( ; ; ) {                    /* do
forever */
        Process_1();
        Process_2();
        Process_3();
        Process_3();
        }
```

لنأخذ على سبيل المثال برنامج لعبة فيديو بسيطة تتألف من مدفع يتحرك لليمين واليسار في أسفل الشاشة ويطلق الصواريخ على سفن فضاءٍ تتحرك أعلى الشاشة.

المهام الأساسية التي يجب أن ينفذها البرنامج هي تلبية أزرار لوحة المفاتيح وتحريك السفن الأعداء واختبار حدوث تصادم (بين الصاروخ وسفينة فضاء أو بين صواريخ الأعداء والمدفع) وتحديث الشاشة.

وبالتالي، يمكن أن يكون الشكل العام للبرنامج كما يلي:

```
for ( ; ; ) {                    /* do
forever */
        check_for_keypressed();
        move_aliens();
        check_for_keypressed();
        check_for_collision();
        check_for_keypressed();
        update_screen();
        }
```

لاحظ أن الإجراء **check_for_keypressed()** الذي يخدم لوحة المفاتيح قد استدعي ثلاثة مرات ضمن الحلقة، وهو بالتالي ينفذ بمعدل أكبر بثلاثة مرات من باقي الإجراءات. والسبب في ذلك هو أنه من الضروري أن يستجيب النظام بسرعة لضغطات اللاعب على لوحة المفاتيح. إذا كانت الإجراءات السابقة قصيرة نسبياً ومتجانسة الطول، نحصل على انطباع أنها تعمل على التفرع دون الحاجة للمقاطعات **interrupts**.

لكن هذا الأسلوب مناسب فقط لنظم الزمن الحقيقي البسيطة فقط بسبب صعوبة تقسيم النظام إلى إجراءات متقاربة الطول بالإضافة إلى أزمان التأخير التي يمكن أن تصبح طويلةً في بعض الحالات.

البرمجة المقادة بالحالة State-Driven Coding

الفكرة في هذا الأسلوب هي تقسيم الإجراء إلى عدة أقسام كل منها يقابل حالة **state** ما. ثم يبرمج الإجراء باستعمال بتعليمات **if-else** متداخلة أو تعليمات **case** أو أية طريقة أخرى تعبر عن أوتومات منتهي **Finite State Automata (FSA)**.

على سبيل المثال، تخيل الإجراء الذي يحقق بروتوكول اتصال ما (مثل **FTP** أو **POP3** ...) من ناحية المخدم. يمكن تقسيم هذا الإجراء بالطريقة المذكورة أعلاه بسهولة كما يلي:

- إذا كنا في حالة انتظار لورود اتصال، وورد اتصال، نرسل رسالة للزبون تسأله عن اسم المستخدم وكلمة المرور ونصبح في حالة انتظار لاسم المستخدم وكلمة المرور.
- إذا كنا في حالة انتظار لاسم المستخدم وكلمة المرور، وورد للمخدم اسم مستخدم وكلمة مرور صحيحتان، نرسل للزبون رسالة ترحيب وننتقل لحالة انتظار لأوامر المستخدم.
- إذا كنا في حالة انتظار لاسم المستخدم وكلمة المرور، وورد للمخدم اسم مستخدم وكلمة مرور غير صحيحتين، نرسل للزبون رسالة تطلب منه إعادة المحاولة ونبقى في نفس حالة انتظار اسم المستخدم وكلمة المرور (مع زيادة عداد المحاولات الفاشلة) وهكذا. يمكن الاستمرار بهذا الشكل لتوصيف كامل إجراء المخدم باستعمال شروط وحالات.

تسمح طريقة التقسيم هذه بتجميد إجراء ما لفترة مؤقتة دون الإخلال بعمله (وذلك بتجميده بعد الانتهاء من تنفيذ تعليمة **if** الموافقة للحالة الحالية). يسهل هذا الأمر عملية تحقيق تعددية المهام بأساليب مثل "الروتينات التعاونية" **Co-routines** التي سنراها في الفقرة التالية. وكذلك، تناسب طريقة التقسيم هذه أسلوب "التنفيذي الدوري" عندما تكون الإجراءات طويلة جداً أو متفاوتة الأطوال.

من ميزات هذه الطريقة أيضاً أنها مبنية على فكرة الأوتومات المنتهي **FSA**. توجد طرق رياضية خوارزمية لأمثلة **optimization** الأوتومات، وهناك أيضاً العديد من الدراسات والنظريات التي يمكن استعمالها للحصول على أفضل تصميم للنظام.

لكن، ليست كل الإجراءات مناسبةً لهذه الطريقة، حيث توجد بعض الإجراءات لا تسمح طبيعتها بأن تقسم لحالات. كذلك، يمكن أن يكون عدد الحالات كبيراً جداً مع وجود علاقات بينها، مما يعقّد كثيراً عملية تحقيقها برمجياً.

الروتينات التعاونية Co-Routines

وتسمى أيضاً "تعددية المهام التعاونية" **cooperative multitasking** وتحتاج لبرمجة منضبطة وتطبيق مناسب لكي تعمل بنجاح.

تتألف النواة التي تعمل بهذا الأسلوب من إجرائين أو أكثر مكتوبين بأسلوب البرمجة المقادة بالحالة المشروحة في الفقرة السابقة. بعد الانتهاء من تنفيذ الحالة الفعالة من الإجراء الفعال، يستدعى موزع **dispatcher** مركزي الذي يقوم باختيار الإجراء التالي الذي سيتم استدعاؤه. وبعد الانتهاء من تنفيذ الحالة الفعالة من هذا الإجراء، يستدعى الموزع من جديد وهكذا.

تتصل الإجراءات فيما بينها باستعمال المتحولات العامة **global variables**، وأية معلومات يحتاج الإجراء للحفاظ عليها من انتقاء لآخر (من قِبل الموزع) يجب كذلك أن يخزن في متحولات عامة.

لنأخذ، على سبيل المثال، نظاماً تعاونياً مكوناً من إجرائين يعملان "على التفرع" وبشكلٍ مستقل:

بعد تنفيذ المرحلة **phase_a1** من الإجراء **process_a**، ينتقل التنفيذ إلى الموزع المركزي (بسبب تعليمة **return**) الذي يستدعي **process_b** الذي يستدعي بدوره **phase_b1**. بعد الانتهاء من **phase_b1**، ينتقل التنفيذ للموزع من جديد الذي يستدعي **process_a** الذي يستدعي بدوره **phase_a2** وهكذا ...

يبين البرنامج التالي الشكل العام للإجراءات **process_a** و **process_b** بلغة **C**: (حلقة الموزع لا تظهر في هذا البرنامج، لكنها بديهية)

```
void process_a ( ) {
        switch(state_a) {
            case 1: phase_a1( );
                    return;
            case 2: phase_a2( );
                    return;
            case 3: phase_a3( );
                    return;
            case 4: phase_a4( );
                    return;
            case 5: phase_a5( );
                    return;
        }
}

void process_b ( ) {
        switch(state_b) {
            case 1: phase_b1( );
                    return;
            case 2: phase_b2( );
                    return;
            case 3: phase_b3( );
                    return;
            case 4: phase_b4( );
                    return;
            case 5: phase_b5( );
                    return;
        }
}
```

المتحولات **state_a** و **state_b** هي متحولات حالة عامة لكلّ من الإجرائين **process_a** و **process_b** على الترتيب.

يمكن بسهولة تعميم هذا البرنامج لجدولة **n** إجراء. ويمكن كذلك تعديل هذا الأسلوب ليعمل مع إجراء ذي حلقة استقصاء. بذلك يمكن للمعالج القيام بأعمال أخرى مفيدة (أي تنفيذ باقي الإجراءات) بدلاً من إضاعة وقته بانتظار حدوث الحدث. يجب طبعاً في هذه الحالة أن تتحول حلقة الاستقصاء إلى فحص للعلام ثم الانتقال للموزع.

باختصار، يعتبر أسلوب الروتينات التعاونية من أسهل خوارزميات الجدولة "العادلة" **fair** التي يمكن تحقيقها بسهولة. كذلك يمكن لعدة مبرمجين أن يتشاركوا بكتابة الإجراءات وليس من الضروري معرفة عدد الإجراءات مسبقاً. أخيراً، تملك بعض لغات البرمجة (مثل **ADA**) بنى جاهزة لتحقيق الروتينات التعاونية بسهولة.

هناك العديد من نظم الزمن الحقيقي الضخمة التي جرى تحقيقها باستعمال الروتينات التعاونية، مثل نظام **Customer Information Control System (CICS)** الذي طورته شركة **IBM** لمعالجة المناقلات **transaction** التي يقوم بها الزبائن، ونظام **OS/2 Presentation Manager** المطور من نفس الشركة.

للأسف لا يمكن استعمال هذا الأسلوب إلا مع الإجراءات التي يمكنها أن تترك المعالج بفواصل منتظمة والتي يمكن تقسيمها لأجزاء منتظمة. كذلك فإن طريقة الاتصال الوحيدة المتاحة بين الإجراءات هي باستعمال المتحولات العامة، وهو أمر غير مرغوب به عموماً.

النظم المقادة بالمقاطعة Interrupt-Driven Systems

في هذا النوع من النظم، يتألف البرنامج الرئيسي من حلقة فارغة (لا تحوي تعليمات) غير منتهية، بالإضافة إلى الإجراءات المختلفة التي تجدول باستعمال مقاطعات مادية **hardware** أو برمجية **software**. تجري عملية التوزيع **dispatching** ضمن روتين تخديم المقاطعة.

عند استعمال الجدولة بالمقاطعة المادية، تولّد ساعة خارجية أو أي جهاز خارجي إشارات مقاطعة توجه إلى متحكم مقاطعات مخصص **interrupt controller**، الذي يقوم بدوره بتوجيهها إلى المعالج حسب ترتيب ورودها وأولوياتها. إذا كان التصميم المادي للنظام (المعالج بشكل أساسي) يدعم عدة مقاطعات، عندها يقوم العتاد المادي أيضاً بعملية التوزيع (أي استدعاء الإجراء الموافق للمقاطعة). أما إذا دعم المعالج مقاطعةً واحدة فقط، عندها تصبح مسؤولية التوزيع ملقاةً على عاتق روتين تخديم المقاطعة وذلك بسؤال متحكم المقاطعات عن مصدر

المقاطعة ثم القفز إلى الإجراء المسؤول عن تلبيتها.

سنستعرض فيما يلي:

- روتينات تخديم المقاطعات **Interrupt Service Routines (ISR)**.
- تبديل السياق **context switch**.

روتينات تخديم المقاطعات Interrupt Service Routines (ISR)

من المهم فهم كيفية عمل المقاطعات عند كتابة برمجيات النظم المضمنة، لأنه غالباً ما يتطلب برنامج الزمن الحقيقي تلبية مقاطعات تجهيزات النظام المختلفة.

إما أن يقوم مهندس البرمجيات بكتابة سواقة الجهاز **device driver** من الصفر (وهي هنا تلعب دور روتينات تخديم المقاطعة)، أو يقوم بتعديل وتخصيص سواقة جهاز عامة **generic** بما يتناسب مع خصوصية الجهاز المادي المكتوبة لأجله.

يوجد نوعان للمقاطعة في أي نظام:

- **مقاطعات مادية:** وهي إشارات تولّدها التجهيزات المحيطية **peripherals** وترسل إلى المعالج الذي يستجيب لها بتنفيذ روتين تخديم المقاطعة **ISR**. يتولى هذا الروتين (وهو برنامج جزئي عادي) بالقيام بما يلزم لتلبية هذه المقاطعة. هذه المقاطعات هي الوسيلة التي تمكّن التجهيزات المحيطية من إخبار المعالج بحاجتها لأن يلتفت إليها لسبب ما (حدوث خطأ، إنتهاء عملية دخل/خرج، ...).
- **مقاطعات برمجية:** تشبه المقاطعات المادية، لكن مصدر المقاطعة هو تعليمات خاصة تُستدعى برمجياً. عادة ما يستّعمل هذا النوع من المقاطعات لاستدعاء الخدمات التي يقدمها نظام التشغيل (التي تكون على شكل روتينات تخديم هذه المقاطعات البرمجية).

ويوجد في لغات البرمجة الحديثة مفهوم هام هو "الاستثناءات" **exceptions** التي تعتبر بمثابة مقاطعات برمجية داخلية (أي يولّدها المعالج داخلياً) عندما يحاول البرنامج تنفيذ عمليات غير قانونية وغير متوقعة (مثل التقسيم على صفر أو محاولة فتح ملف غير موجود). يقفز المعالج عند حدوث الاستثناء إلى روتين تخديم المقاطعة وينفّذ التعليمات الموافقة له.
توصف المقاطعات المادية بأنها ذات طبيعة غير متزامنة، أي أنها يمكن أن تحدث في أية لحظة. وعندما تحدث، يتوقف تنفيذ البرنامج الذي كان المعالج ينفذه وينتقل التنفيذ (فوراً أو بعد حين) إلى روتين تخديم المقاطعة **ISR**.

يحتاج مطور النظام عادةً لكتابة **ISR** لبعض المقاطعات المادية، ومن المهم عندها أن يفهم جيداً مم تتألف حالة المعالج وما هي السجلات التي يجب على **ISR** أن يحفظها.

عندما يتشارك أي برنامج مع روتين تخديم مقاطعةٍ ما ببعض المصادر المشتركة، فإن الطريقة الوحيدة لتنسيق عملية الوصول إليها هي حجب المقاطعة قبل الوصول إليها في البرنامج وتأهيلها من جديد بعد الانتهاء منها.

السبب هو أنه لا يمكن استعمال طرق التزامن (أو المنع المتبادل **mutual exclusion**) الأخرى داخل روتين تخديم المقاطعة لأن ذلك يمكن أن يسبب تجميده لفترة طويلة وغير محدودة بانتظار تحرير المصدر، وهذا أمر غير مرغوب فيه لأنه يمكن أن يؤخر تلبية مقاطعاتٍ هامةٍ لفترة غير محدودة. وعندما تكون المقاطعات محجوبة، تصبح استجابة النظام للأحداث الخارجية ضعيفة جداً.

لذلك يجب أن يكون الجزء من البرنامج الذي يحجب المقاطعات أقصر ما يمكن.

نقول عن **ISR** أنه "قابل لإعادة الدخول" **reentrant** إذا سمح بتلبية أي مقاطعة قبل أن ينتهي تنفيذه (حتى ولو كانت المقاطعة الجديدة هي نفسها التي يقوم بتلبيتها حالياً لكن حدثت لسبب آخر).

مهما كان نوع المقاطعة (مادية أو برمجية)، يجب على **ISR** تخزين حالة النظام (وهو ما يسمى بالسياق **context**) قبل أن يقوم بعمله حتى يكون بالإمكان متابعة تنفيذ الإجراء المقاطَع بعد الانتهاء من تلبية المقاطعة.

تبديل السياق context switch

نعرف تبديل السياق بأنه عملية حفظ معلومات كافية (سياق) عن إجراء زمن حقيقي نتيجة مقاطعته، بحيث يمكن متابعة تنفيذه لاحقاً بعد الانتهاء من تنفيذ روتين تخديم المقاطعة.

ويستعمل هذا المصطلح أيضاً عندما ينقل الموزع التنفيذ من إجراء لآخر: تحفظ معلومات الأول وتُسترجع معلومات الثاني ليتابع التنفيذ بدلاً من الأول.

تُحفَظ معلومات السياق عادةً على مكدس النظام **system stack**.

ويؤثر الزمن اللازم لتبديل السياق تأثيراً كبيراً على زمن استجابة النظام، لذلك فإنه يجب أن يكون أمثلياً وأصغر ما يمكن.

عادةً ما تتضمن معلومات السياق الأمور التالية:

- محتويات السجلات العامة للمعالج لحظة المقاطعة.
- قيمة سجل عداد البرنامج **program counter**.
- قيم سجلات المعالج الحسابي المساعد **math coprocessor** (في حال وجوده).
- سجل صفحة الذاكرة.
- مواقع الذاكرة التي تعنون تجهيزات دخل/خرج **memory-mapped I/O**.

وعادةً، يقوم روتين تخديم المقاطعة بتحجيب المقاطعات أثناء قيامه بتبديل السياق لأنها تُعتبر عمليةً حرجة يجب أن تتم دون مقاطعة، ثم يفعلها من جديد بعد الانتهاء من التبديل.

يستعمل أسلوب تخزين السياق على المكدس في النظم المضمنة على الأغلب، لأن عدد إجراءات الزمن الحقيقي أو الإجراءات المقادة بالأحداث محدد في هذه النظم. عندما تحدث المقاطعة، يستدعي المعالج روتين تخديم المقاطعة المناسب الذي يقوم بحفظ سياق الإجراء قيد التنفيذ على المكدس (أو في منطقة محددة وساكنة في الذاكرة إذا كان النظام وحيد المقاطعة).

يبين شبه البرنامج التالي جزءاً من نظام زمن حقيقي مكتوب بلغة **C**. ويتكون من برنامج رئيسي يحوي حلقة **while** فارغة وغير منتهية، وثلاثة روتينات تخديم مقاطعة تستعمل أسلوب تخزين السياق على مكدس.

```
void main( )
/* initialize system, load interrupt handlers */
{
        init( );
        while(TRUE);        /* infinite wait loop */
}

void int1 ( )               /* interrupt handler 1 */
{
        save(context);      /* save context on stack */
        task1( );           /* execute task 1 */
        restore(context); /* restore context from stack */
}
```

```
void int2( )                /* interrupt handler 2 */
{
        save(context);      /* save context on stack */
        task2( );           /* execute task 2 */
        restore(context);   /* restore context from stack */
}

void int3( )                /* interrupt handler 3 */
{
        save(context);      /* save context on stack */
        task3( );           /* execute task 3 */
        restore(context);   /* restore context from stack */
}
```

يجب على الإجرائية **init()** أن تحفظ عناوين التوابع الثلاث **int1** و **int2** و **int3** في مكان محدد من الذاكرة اسمه شعاع المقاطعة **interrupt vector**. عندما ترد مقاطعة إلى المعالج، يستفهم المعالج من متحكم المقاطعات عن مصدرها كما ذكرنا سابقاً، ثم يستفيد من شعاع المقاطعة للقفز إلى روتين تخديم المقاطعة الموافق لها.

تقوم الإجرائية **save()** بحفظ قيم السجلات الهامة على المكدس، بينما تقوم الإجرائية **restore()** باسترجاعها منه.

نظم الأولوية الشُفعية Preemptive-Priority

نقول عن إجراء ذو أولوية أعلى أنه يقوم بشُفع (اغتصاب، سرقة) المعالج من الإجراء قيد التنفيذ والذي أولويته أقل إذا قاطعه وجعل المعالج ينفذ تعليماته.

تُسمى النظم التي تستعمل أسلوب الجدولة هذا بدلاً من الأساليب الأخرى (الشريط الدوار **round-robin** أو القادم أولاً يخدم أولاً **first come first serviced**) بنظم الأولوية الشفعية.

تعتمد أولويات المقاطعات على أهمية وإلحاح المهمات المناطة بها.

على سبيل المثال، الطريقة الأمثل لتحقيق نظام مراقبة بمحطة توليد كهرباء نووية هي طريقة الأولوية الشفعية. صحيح أن التعامل مع دخيل على المحطة هو أمر مهم، لكن لا شيء أهم من التعامل مع إنذار ارتفاع درجة حرارة نواة المفاعل، وله بالتالي الأولوية الأكبر.

يمكن أن تكون أولويات المقاطعات ساكنةً أو ديناميكية (قابلة للتغيير). النظم ذات المقاطعات الساكنة أقل مرونةً لأنه لا يمكن تغيير أولويات المقاطعات أثناء عمل النظام بما يتناسب مع تغير احتياجاته، بينما يمكن عمل ذلك في النظم ذات المقاطعات الديناميكية.

وتعاني نظم الأولوية الشفعية من مشكلة استئثار الإجراءات ذات الأولوية الأعلى بالمصادر المتاحة في النظام، مما يؤدي لحرمان الإجراءات ذات الأولوية الأدنى منها. نقول في هذه الحالة أن الإجراءات ذات الأولوية الأقل تعاني من مشكلة "المجاعة" **starvation**.

هناك نوع خاص من نظم الأولوية الشفعية المقادة بالمقاطعات تسمى نظم "الرتيبة المعدل" **Rate Monotonic**، وهي تضم نظم الزمن الحقيقي ذات الإجراءات الدورية التي يعطَى فيها الأولوية الأعلى للإجراء الدوري الذي له أعلى معدل تنفيذ (أي ينفذ عدداً أكبر من المرات في الثانية).

هذا الأسلوب شائع في النظم المضمنة وخصوصاً نظم الطيران، وهناك الكثير من الدراسات حوله.

لنأخذ على سبيل المثال نظام الملاحة في الطائرة. الإجراء الذي يحصل المعطيات من جهاز قياس التسارع كل **10** ميلي ثانية له أعلى أولوية. يأتي في الدرجة الثانية الإجراء الذي يحصل معطيات التوازن كل **40** ميلي ثانية. وأما الإجراء الذي يظهر المعطيات على شاشة الطيار كل ثانية فإن له الأولوية الأدنى.

سندرس النظم الرتيبة المعدل في فقراتٍ لاحقة بالتفصيل.

النظم الهجينة Hybrid Systems

تحتوي هذه النظم على مقاطعات تحدث دورياً تعبر عن الإجراءات الدورية **periodic**، بالإضافة إلى مقاطعات غير دورية تعبر عن الإجراءات الغير دورية **aperiodic** أو المتباعدة **sporadic**.

يمكن استعمال المقاطعات الغير دورية لمعالجة الأخطاء الحرجة التي تحدث استثنائياً ويجب معالجتها فوراً، وبالتالي يكون لها الأولوية الأعلى.
وعادةً ما نجد هذا الخليط من المقاطعات في النظم المضمنة أيضاً.

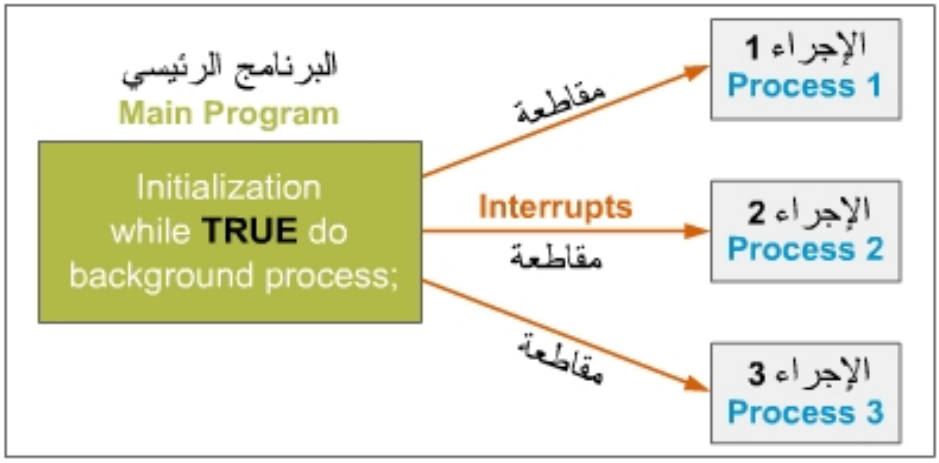
هناك نوع آخر من النظم الهجينة موجود في نظم التشغيل التجارية وهو مزيج من النظم الشُفعية مع أسلوب الشريط الدوار **round-robin**. في هذه النظم، يمكن دائماً للإجراءات ذات الأولوية الأعلى أن تقوم بشُفع **preempt** الإجراءات ذات الأولوية الأدنى. لكن إذا وجد المجدول عدة إجراءات لها نفس الأولوية وجاهزة للتنفيذ، يقوم بجدولتها بطريقة الشريط الدوار التي سندرسها لاحقاً.

نظم الواجهة الأمامية/الواجهة الخلفية Foreground/Background

هذا النوع من النظم هو تحسين للنظم المقادة بالمقاطعة، حيث تُستبدل حلقة الاستقصاء التي لا تقوم بأعمال مفيدة بحلقة تؤدي عمليات معالجة مفيدة بانتظار حدوث الحدث، وهو النوع الأكثر استعمالاً لتحقيق النظم المضمنة.

تتألف هذه النظم من عدد من إجراءات الزمن الحقيقي المقادة بالمقاطعة (وتسمى "الواجهة الأمامية")، بالإضافة إلى عدد من الإجراءات الغير مقادة بالمقاطعة (وتسمى "الواجهة الخلفية").

ويبين الشكل التالي نظاماً من هذا النوع:



تعمل إجراءات الواجهة الأمامية بطريقة الشريط الدوار أو الأولوية الشُفعية أو النظم الهجينة. أما إجراء الواجهة الخلفية فهو قابل لأن يشفع من قِبل أي إجراء من الواجهة الأمامية. لذلك فهو يمثّل الإجراء ذو الأولوية الأدنى في النظام.

وكل أنظمة الزمن الحقيقي هي حالة خاصة من هذا النوع من النظم.

على سبيل المثال، حلقة الاستقصاء هو نظام واجهة أمامية/واجهة خلفية دون واجهة أمامية (تمثّل حلقة الاستقصاء الواجهة الخلفية). وإذا أضفنا المقاطعات، نحصل على نظام واجهة أمامية/واجهة خلفية كامل.

وإن النظم المقادة بالحالة هي أيضاً نظم واجهة أمامية/واجهة خلفية ليس لها واجهة أمامية وحلقتها المقادة بالحالة هي واجهتها الخلفية. والروتينات التعاونية هي أيضاً حالة خاصة فيها إجراء خلفي معقّد فقط.

أخيراً، النظم المقادة بالمقاطعة هي أيضاً نظم واجهة أمامية/واجهة خلفية ليس لها واجهة خلفية.

المعالجة في الواجهة الخلفية

بما أن الواجهة الخلفية هي إجراء غير مقاطِعٍ بالمقاطعة، فهي تحتوي على المعالجة الغير حرجة زمنياً في النظام. صحيح أن إجراء الواجهة الخلفية له أدنى أولوية، لكنه ينفذ دوماً إذا كانت انشغالية النظام أصغر تماماً من **100%** ولايحوي منعاً متبادلاً **deadlock**. على سبيل المثال، يمكن زيادة عداد في الواجهة الخلفية لإعطاء قياسٍ عن مدى انشغالية النظام أو لاكتشاف حالة دخول أحد إجراءات الواجهة الأمامية بحلقة غير منتهية (عندها لا تحدث أية زيادة في العداد لفترة طويلة). يمكن أيضاً أن تحتفظ الواجهة الخلفية بعداد لكل إجراء واجهة أمامية يصفر في بداية تنفيذ الإجراء الموافق. إذا وجد إجراء الواجهة الخلفية أن هناك عداد لا يصفر بالمعدل الكافي، فهذا يدل على أن الإجراء الموافق له لا ينفذ بالمعدل المطلوب وأن هناك خطأ ما في النظام. يشبه هذا الأمر "مؤقت كلاب الحراسة" **watchdog timer** المتاح مادياً في بعض الأنظمة، ولكنه برمجي بالطريقة التي عرضناه بها.

يمكن أيضاً وضع عمليات اختبار ذاتي منخفضة الأولوية في إجراء الواجهة الخلفية. على سبيل المثال، يمكن في الكثير من الأنظمة إجراء اختبار كامل لمجموعة تعليمات المعالج في الواجهة الخلفية. هذا الأمر يجعل النظام متيناً وموثوقاً جداً. أخيراً، يمكن أن نضع في الواجهة الخلفية عمليات تحديث لشاشات الإظهار المنخفضة الأولوية أو الطباعة على طابعة أو التخاطب مع أية تجهيزات دخل/خرج بطيئة ومنخفضة الأولوية.

التهيئة initialization

التهيئة هي الجزء الأول من إجراء الواجهة الخلفية، وتتضمن الخطوات التالية:

- تحجيب المقاطعات.
- تهيئة شعاع المقاطعة والمكدس.
- إجراء اختبارٍ ذاتي للنظام.
- عمليات تهيئةٍ أخرى للنظام.
- تأهيل المقاطعة من جديد.

من المهم أن تُحجب المقاطعات في بداية عملية التهيئة، لأن بعض النظم تقلع بمقاطعات مؤهلة بينما لم تجهز الأمور الضرورية لاستقبال المقاطعات بعد، مثل شعاع المقاطعة والمكدس. من المهم كذلك إجراء اختبار ذاتي للنظام قبل تفعيل المقاطعات. بعدها، يصبح النظام جاهزاً للعمل.

عمليات الزمن الحقيقي

إن عمليات الزمن الحقيقي في الواجهة الأمامية في نظام واجهة أمامية/واجهة خلفية تقابل تماماً المقاطعات في النظم المقادة بالمقاطعة.

لنأخذ على سبيل المثال حالة نظام فيه مقاطعة وحيدة ويحتوي على إجراء واجهة أمامية واحد وإجراء الواجهة الخلفية. نفرض أن تعليمة **EPI** تفعل المقاطعة وأن **DPI** تحجبها وأن المعالج يحجب المقاطعة آلياً عند حدوثها (ضمنياً دون استعمال **DPI**) إلى أن تُفعل صراحةً باستدعاء **EPI**.

من أجل تحقيق عملية تبديل السياق، يجب حفظ قيم سجلات المعالج الثمانية **R1-R8** على المكدس من قِبل إجراء الواجهة الأمامية وذلك لأن إجراء الواجهة الخلفية يستعملها ولا نريد أن تضيع قيمها عند حدوث المقاطعة. سيعمل إجراء الواجهة الأمامية حتى ينتهي، لذلك لا حاجة لحفظ سياقه لأنه لن يقاطَع. نفترض أخيراً أن عنوان روتين تخديم المقاطعة مخزن في العنوان **5** من الذاكرة (وهو شعاع المقاطعة).

وتؤدي تعليمات التجميع التالية مهمة تهيئة هذا النظام البسيط:

```
DPI                 ; disable interrupts
STORE &handler,5 ; put interrupt handler address in location 5
                    ; other initializations can be done here
EPI                 ; enable interrupts
```

إذا فرضنا أن **reg0** حتى **reg7** تمثّل عناوين في الذاكرة نستعملها لحفظ قيم سجلات المعالج، عندها يمكن أن نكتب روتين تخديم المقاطعة كما يلي:

```
DPI                 ; redundantly disable interrupts, not required
STORE R0,&reg0 ; save register 0
STORE R1,&reg1 ; save register 1
STORE R2,&reg2 ; save register 2
STORE R3,&reg3 ; save register 3
STORE R4,&reg4 ; save register 4
STORE R5,&reg5 ; save register 5
STORE R6,&reg6 ; save register 6
STORE R7,&reg7 ; save register 7
JU @APP             ; execute real-time application program
```

```
LOAD R7,&reg7   ; restore register 7
LOAD R6,&reg6   ; restore register 6
LOAD R5,&reg5   ; restore register 5
LOAD R4,&reg4   ; restore register 4
LOAD R3,&reg3   ; restore register 3
LOAD R2,&reg2   ; restore register 2
LOAD R1,&reg1   ; restore register 1
LOAD R0,&reg0   ; restore register 0

EPI             ; re-enable interrupts
RI              ; return from interrupt
```

توجد في العديد من المعالجات تعليمات خاصة مثل **save** و **restore** تقوم بحفظ واسترجاع قيم جميع سجلات المعالج الهامة في مواقع متتالية من الذاكرة. لاحظ أن روتين المقاطعة هذا لا يسمح بأن يقاطَع لأنه لا يعيد تفعيل المقاطعة حتى نهايته. إذا أردنا أن نسمح بمقاطعة روتين المقاطعة، فإنه يجب استعمال مكدس لتخزين قيم سجلات المعالج بدلاً من المواقع الساكنة **reg0** حتى **reg7**.

ويمكن لإجراء الواجهة الخلفية أن يشمل استدعاءاً لتابع التهيئة، بالإضافة لأية عمليات معالجة غير حرجة زمنياً. على سبيل المثال، يبين البرنامج التالي المكتوب بلغة **C** مثل هذا الإجراء:

```
void main ( )
/*allocate space for context variable */
int reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7;
/*declare other global variables here */
{
        init();                 /*initialize system */
        while (TRUE)            /*background loop */
                background(); /* non-real-time processing here */
}
```

تتميز نظم الواجهة الأمامية/الواجهة الخلفية عادةً بزمن استجابة جيد لأنها تعتمد على العتاد المادي لجدولة إجراءاتها، وهي الخيار المفضل لدى الكثير من مصممي نظم الزمن الحقيقي المضمنة.

لكن توجد فيها نقطة ضعف أساسية وهي ضرورة كتابة الواجهات البرمجية المعقدة (سواقات الأجهزة) للكثير من الأجهزة المحيطية يدوياً. هذا الأمر صعب ومليء بالأخطاء والمشاكل.

بالإضافة إلى ذلك، يناسب هذا النوع من النظم التطبيقات التي يكون فيها عدد إجراءات الواجهة الأمامية ثابتاً ومعروفاً مسبقاً. لذلك يصعب استعمالها في التطبيقات التي يتغير فيها عدد

إجراءات الواجهة الأمامية. وعلى الرغم من أن اللغات التي تدعم الحجز الديناميكي للذاكرة يمكنها التعامل مع عدد متغير من الإجراءات، لكن يبقى الأمر غير سهل.

نظم تشغيل الزمن الحقيقي كاملة المواصفات

يمكن تمديد نموذج الواجهة الأمامية/الواجهة الخلفية ليصبح نظام تشغيل حقيقي كامل المواصفات بإضافة مهام أساسية في أي نظام تشغيل مثل واجهات شبكة وسواقات أجهزة وأدوات تنقيح **debugging** متقدمة وغيرها ...

وهناك العديد من النظم التجارية الجاهزة المبنية بهذا الأسلوب.

نموذج كتلة تحكم الإجراء Task Control Block (TCB)

هذا النموذج هو الأكثر استعمالاً في نظم تشغيل الزمن الحقيقي التجارية الكاملة المواصفات لأنه يسمح بأن يكون عدد الإجراءات متغيراً. وهو يستعمل أيضاً في النظم التفاعلية المتصلة **on-line** حيث يزيد عدد الزبائن المتصلين بالنظام وينقص باستمرار، ويخصص لكل زبون متصل إجراء مستقل.

ويمكن استعمال عدة خوارزميات جدولة مع هذا النموذج، مثل الشريط الدوار والأولوية الشُفعية أو أية تركيبة منهما، على الرغم من أنه عادةً ما تُستعمل خوارزمية الشريط الدوار معه.

المشكلة الوحيدة في نموذج **TCB** هي التأخير الزمني الكبير الذي تتطلبه عملية الجدولة إذا كان عدد الإجراءات المنشأة كبيراً.

ينشئ النظام في هذا النموذج بنية معطيات خاصة (تسجيلة **record**) لكل إجراء يعمل في الذاكرة. وتسمى هذه التسجيلة "كتلة تحكم الإجراء" **Task Control Block**، وتحتوي على الأقل على المعلومات التالية:

- قيمة سجل عداد البرنامج **PC**.
- قيم سجلات المعالج في اللحظة التي قوطع فيها هذا الإجراء (السياق).
- رقم أو اسم تعريف للإجراء.
- حالة الإجراء (جاهز، متوقف، في حالة سبات، ...).
- أولوية الإجراء (في حال استعمال مفهوم الأولوية).

ويخزن النظام كتل التحكم هذه في بنية معطيات مناسبة (لائحة مترابطة **linked-list** على

الأغلب).

سنستعرض فيما يلي:

- حالات الإجراء.
- إدارة الإجراءات.

إدارة المصادر

حالات الإجراء

يدير نظام التشغيل الإجراءات بحفظ حالة كل منها في كتلة تحكمه. وعادةً ما يكون الإجراء في إحدى الحالات الأربع التالية:

- قيد التنفيذ **executing**.
- جاهز للتنفيذ **ready**.
- متوقف **suspended** (أو مجمد **blocked**).
- في سبات **dormant** (أو نائم **sleeping**).

الإجراء قيد التنفيذ هو الإجراء الذي ينفّذه المعالج في هذه اللحظة، وهو وحيد إذا احتوى النظام على معالج واحد فقط. إذا كان النظام متعدد المعالجات، يمكن لعدة إجراءات أن تكون في حالة "قيد التنفيذ".

ويمكن لإجراء أن يدخل هذه حالة بعيد إنشائه (إذا لم يكن هناك أي إجراء آخر في حالة "جاهز للتنفيذ")، أو من حالة "جاهز للتنفيذ" (إذا كان مؤهلاً للتنفيذ حسب أولويته أو موقعه في رتل الشريط الدوار للإجراءات الجاهزة للتنفيذ).

عندما ينتهي تنفيذ الإجراء، ينتقل إلى الحالة "متوقف".
الإجراءات التي حالتها "جاهز للتنفيذ" هي تلك الإجراءات التي يمكن أن تنتقل لحالة "قيد التنفيذ" لكنها ليست "قيد التنفيذ".

ويدخل الإجراء حالة "جاهز للتنفيذ" إذا كان "قيد التنفيذ" وانتهت حصته الزمنية التي خصصها المجدول له، أو إذا جرى شفعها **preempted** من قِبل إجراء آخر له أولوية أعلى. كذلك، إذا كان الإجراء في حالة "متوقف"، وحدث الحدث الذي ينتظره، تصبح حالة الإجراء "جاهز للتنفيذ".

الإجراءات في حالة "متوقف" أو "مجمد" هي تلك الإجراءات التي تنتظر توفر مصدر محدد، فهي

بالتالي ليست "جاهزة للتنفيذ". عندما يتوفر هذا المصدر، ينتقل الإجراء لحالة "جاهز للتنفيذ".
تُستعمل حالة السبات في النظم التي يكون فيها عدد كتل التحكم محدوداً.

ويمكن توصيف هذه الحالة بأنها حالة إجراء موجود لكنه غير متاح لمجدول نظام التشغيل، لامتلاء جدول كتل التحكم.
عند إنشاء إجراء جديد، يمكن أن يوضع في حالة سبات إلى أن يتوقف أحد الإجراءات الأخرى ويصبح مكانه متاحاً في جدول كتل التحكم. عندها ينتقل من حالة سبات إلى حالة "جاهز للتنفيذ".

إدارة الإجراءات

يمكن النظر إلى نظام التشغيل على أنه الإجراء ذو الأولوية الأعلى في النظام. وإن أية مقاطعة أو استدعاء لخدمة من خدمات النظام (مثل طلب مصدر ما) تستدعي نظام التشغيل.

يحتفظ نظام التشغيل بلائحتين مترابطتين من كتل التحكم. الأولى للإجراءات الجاهزة والثانية للإجراءات المتوقفة. ويحتفظ كذلك بجدول للمصادر المتاحة وآخر لطلبات المصادر التي طلبتها الإجراءات.

تحتوي كل كتلة إجراء نفس المعلومات التي يحتفظ بها روتين تخديم المقاطعة على المكدس.
ويبين الشكل التالي مثالاً لبنية كتلة تحكم الإجراء:

رقم تعريف الإجراء
الأولوية
حالة الإجراء
السجل 1 ..... السجل n
سجل عداد البرنامج
سجلات حالة المعالج
مؤشر لكتلة التحكم التالية

يتميز نموذج **TCB** بأنه مناسب عندما لا يكون عدد إجراءات النظام محدداً خلال مرحلة تصميمه، أو يمكن أن يتغير عددها أثناء عمله. أي أنه نموذج مرن جداً.

إن الأسلوب الأبسط لعمل المجدول في هذا النموذج هو على الشكل التالي:

عند استدعائه (دورياً أو عند حدوث حدث)، يفحص المجدول اللائحة المترابطة للإجراءات الجاهزة للتنفيذ. إذا وجِد إجراء في هذه اللائحة، عندها تُنقَل كتلة تحكم الإجراء الذي كان قيد التنفيذ إلى آخر لائحة الإجراءات الجاهزة، وينتقي المجدول أحد الإجراءات الجاهزة للتنفيذ لنقله إلى حالة "قيد التنفيذ".

يمكن تبسيط عملية إدارة الإجراءات بتغيير حالتها فقط دون نقلها من لائحة مترابطة لأخرى.

على سبيل المثال، بفرض أن عدد كتل التحكم الأعظمي ثابتاً، يمكن أن يحتفظ النظام بالعدد الأعظمي من كتل التحكم في لائحة واحدة وتكون كلها في حالة سبات عند إقلاعه. يمكن بعدها إضافة إجراءات جديدة للنظام باختيار أحد الكتل التي في حالة سبات وتهيئة حقولها وتغيير حالتها إلى "جاهز للتنفيذ". أثناء عمل النظام، تتغير حالة الإجراءات بين القيم الممكنة لها ("قيد التنفيذ"، "جاهز للتنفيذ"، "متوقف") حسب ما يطرأ عليها من أحداث وتبقى في نفس اللائحة. وعند انتهاء تنفيذ إجراء ما، يزال من النظام بتغيير حالته إلى "سبات".

وميزة هذا الأسلوب أنه يلغي الحاجة إلى الذاكرة الديناميكية التي تتطلبها إدارة كتل التحكم، كما أنه يجعل أداء النظام محدداً وقابلاً للتنبؤ به لأن لائحة كتل التحكم لها طول ثابت.

إدارة المصادر

بالإضافة إلى عملية الجدولة، يهتم نظام التشغيل بمعرفة حالة كل مصدر متاح في النظام (جهاز دخل/خرج، ذاكرة،...) ويحتفظ بمعلومات عنها في جدول خاص اسمه "جدول المصادر".

يفحص النظام دورياً حالة الإجراءات الموجودة في رتل الإجراءات المتوقفة. فإذا وجد إجراءاً متوقفاً لأنه ينتظر توفر مصدر ما، ووجد أن هذا المصدر أصبح متوفراً، عندها يغير حالته إلى "جاهز للتنفيذ" وينقل كتلة تحكمه إلى رتل الإجراءات الجاهزة للتنفيذ.

2. الأسس النظرية لنظم تشغيل الزمن الحقيقي

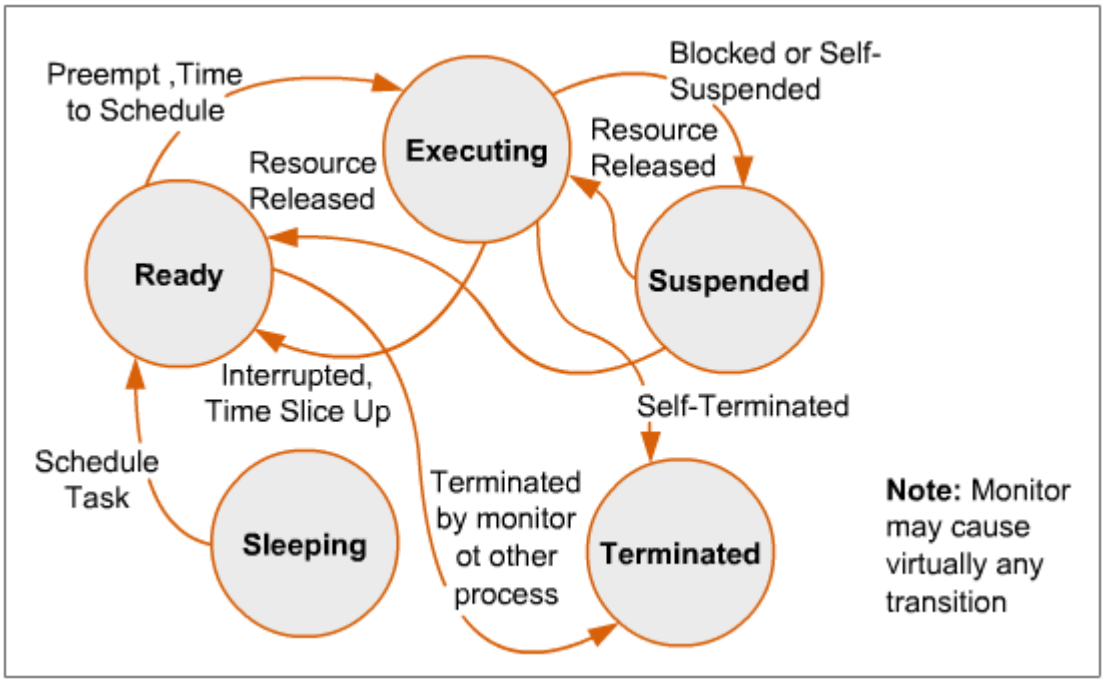
نحتاج في البداية لصياغة بعض المفاهيم النظرية بشكل رياضي وواضح لكي نتمكن من فهم نظم تشغيل الزمن الحقيقي فهماً صحيحاً. نستعرض بعد ذلك خوارزميات جدولة إجراءات الزمن الحقيقي المختلفة ونناقش مزايا ومساوئ كل منها.

تتصف معظم نظم الزمن الحقيقي بأنها تشاركية **concurrent**، أي أن طبيعة تفاعلها مع الأحداث الخارجية تتطلب عادةً أن تقوم عدة إجراءات بالتخاطب مع العالم الخارجي في نفس

الوقت (على التفرع).

وتحدثنا سابقاً عن الإجراءات ورأينا أن لكل منها حالة يغيرها المجدول حسب حدوث أحداث عينة (انتهاء الحصة الزمنية لإجراء، طلب الإجراء لمصدر محجوز من قبل إجراء آخر، ...).

يبين الشكل التالي الحالات المختلفة للإجراءات والأحداث التي تسبب الانتقالات فيما بينها:



تجدر الإشارة أن بعض نظم التشغيل تستعمل مصطلحات مختلفة عن المصطلحات المستعملة هنا، لكن الأفكار العامة تبقى نفسها.

سنستعرض فيما يلي:

- جدولة الإجراءات.
- جدولة الأولويات الثابتة – الخوازمية "الرتيبة المعدل" **Rate Monotonic (RM)**.
- جدولة الأولويات الديناميكية – خوارزمية "الحد الأبكر أولاً" **Earliest-Deadline-First (EDF)**.
- مقارنة الخوارزميتين **RM** و **EDF**.

جدولة الإجراءات

تُعتبر الجدولة من المهام الأساسية لنظام التشغيل. لكي نحقق الاحتياجات الزمنية لبرنامج زمن حقيقي، نحتاج لاستعمال خوارزميات جدولة مختلفة عن تلك المستعملة في نظم التشغيل العادية.

نحتاج كذلك لاسراتيجيات خاصة لترتيب استعمال مصادر النظام من قِبل إجراءات الزمن الحقيقي حسب أولوياتهم وقيودهم الزمنية. هناك نوعين أساسيين لخوارزميات الجدولة في الزمن الحقيقي: خوارزميات ما قبل التنفيذ وخوارزميات زمن التنفيذ. الهدف المشترك لهذه الخوارزميات هو تحقيق الشروط الزمنية لإجراءات الزمن الحقيقي.

تقوم خوارزميات ما قبل التنفيذ بجدولة الإجراءات مسبقاً بحيث تكون الجدولة الناتجة محققة للقيود الزمنية ولا يوجد تضارب في استعمال المصادر (أي لا يستعمل أكثر من إجراء نفس المصدر في نفس اللحظة)، وذلك قبل بدء التنفيذ الفعلي للإجراءات. تحاول هذه الخوارزميات أيضاً تخفيف كلفة تبديل السياق قدر الإمكان، مما يزيد احتمال إيجاد جدولة محققة لجميع القيود الزمنية.

أما في خوارزميات زمن التنفيذ، فتعطى أولويات ساكنة للإجراءات، وتخصص المصادر حسب هذه الأولويات. تعتمد هذه الخوارزميات على تقنيات معقدة في زمن التنفيذ لتحقيق الاتصال والتزامن بين الإجراءات.

العبء على المعالج وصفات المهام

يعرف العبء على المعالج بأنها مجموعة الإجراءات التي يجب على المعالج تنفيذها في لحظة محددة. يمكن لمعالج أن ينفّذ إجراءاً واحداً على الأكثر في لحظة ما، وكل إجراء ينفذ على معالج واحد على الأكثر في لحظة ما. ويوصف كل إجراء زمن حقيقي τ_i بالواصفات الزمنية التالية:

وهناك بعض العلاقات الرياضية التي تربط بين هذه الواصفات الزمنية:

$$\varphi_i = r_{i,1} \text{ and } r_{i,1} = \varphi_i + (k-1) * p_i \quad \textbf{(3.1)}$$

وإذا فرضنا أن $d_{i,j}$ هو الحد الزمني المطلق للتكرار رقم **j** للإجراء τ_i ، يكون لدينا:

$$d_{i,j} = \varphi_i + (j-1) * p_i + D_i \quad \textbf{(3.2)}$$

وإذا كان الحد الزمني النسبي لإجراء دوري τ_i يساوي دوره p_i ، يكون لدينا:

$$d_{i,k} = r_{i,k} + \mathrm{p}_i = \varphi_i + k * p_i \quad \textbf{(3.3)}$$

حيث **k** هو عدد صحيح موجب أكبر أو يساوي **1**، وهو يعبر عن التكرار رقم **k** للإجراء τ_i .

- **قيود الأسبقية precedence constraints:** تحدد فيما إذا كان هناك إجراء (أو إجراءات) يجب أن تُنفذ قبل تنفيذ τ_i .

- **زمن الوصول arrival أو التحرير release:** رمزه $r_{i,j}$ وهو اللحظة التي يحين فيها تنفيذ التكرار رقم j للإجراء τ_i (إذا كان دورياً، أي يكرر تنفيذه كل دور).
- **الطور phase:** رمزه φ_i وهو زمن التحرير لأول تكرار للإجراء τ_i.
- **زمن الاستجابة response time:** وهو الفترة الزمنية بين لحظة تحرير الإجراء ولحظة اكتمال تنفيذه.
- **الحد الزمني المطلق absolute deadline:** رمزه d_i وهو اللحظة الزمنية التي يجب أن ينتهي الإجراء قبلها.
- **الحد الزمني النسبي relative deadline:** رمزه D_i وهي القيمة العظمى المسموحة لزمن الاستجابة.
- **نوعية السماحية laxity type:** هو مقدار الإلحاح **urgency** في تنفيذ الإجراء τ_i.
- **الدور period ورمزه p_i:** هو الفترة الزمنية التي تفصل بين لحظة التحرير لإجراء دوري ولحظة التحرير التالية مباشرة.
- **زمن التنفيذ execution time ورمزه e_i:** وهو الزمن (الأعظمي) اللازم لإتمام تنفيذ الإجراء τ_i عندما ينفذ لوحده وتكون جميع المصادر التي يحتاجها متاحة.

نموذج مبسط للإجراء

سنعتمد نموذجاً مبسطاً للإجراء لكي نتمكن من توصيف بعض خوارزميات الجدولة النظامية المستعملة في نظم الزمن الحقيقي. يفترض هذا النموذج الفرضيات التالية:

- كل الإجراءات الموجودة في النظام دورية.
- الحد الزمني النسبي لكل إجراء يساوي دوره.
- كل الإجراءات مستقلة عن بعضها البعض، أي لا توجد قيود أسبقية بينها.
- لا يحوي أي إجراء قسماً غير شُفعي (أي غير قابل للمقاطعة)، وكلفة تبديل السياق مهملة.
- لكل إجراء احتياجات معالجة فقط (أي يحتاج فقط للمعالج) وتُهمل الاحتياجات الأخرى (ذاكرة، دخل/خرج،...).

من المهم جداً في نظم الزمن الحقيقي أن تكون الجدولة الناتجة عن خوارزمية الجدولة قابلة للتنبؤ **predictable**، أي أنه يمكننا معرفة الإجراء الذي سينفذ لاحقاً في كل لحظة. تستعمل العديد من نظم تشغيل الزمن الحقيقي خوارزمية الشريط الدوار الذي يتميز بأنه بسيط وقابل للتنبؤ.

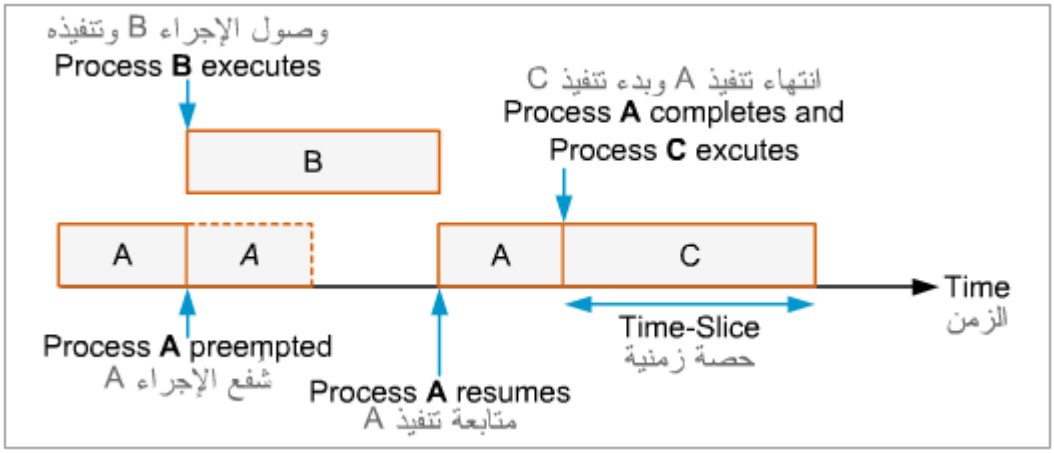
خوارزمية جدولة الشريط الدوار round-robin

في هذه الخوارزمية، يجري تنفيذ الإجراءات تسلسلياً حتى تنتهي وغالباً ما توجد في نظم التنفيذي الدوري **cyclic executive**.

يخصص لكل إجراءٍ شريحة (حصة) زمنية ثابتة **quantum or slice** ينفذ خلالها. تُستعمل ساعة ذات دور ثابت يساوي الحصة الزمنية لمقاطعة النظام دورياً. ويستمر تنفيذ الإجراء حتى ينتهي تنفيذه أو تنتهي حصته الزمنية بحدوث مقاطعة الساعة؛ فإذا لم ينته تنفيذ الإجراء، يحفَظ سياقه ويوضع في نهاية رتل الإجراءات الجاهزة. بعد ذلك، يسترجع النظام سياق الإجراء التالي الموجود في بداية الرتل ويتابع تنفيذه.

إذاً، تعطي هذه الخوارزمية جدولةً عادلةً غير شُفعية للإجراءات التي لها نفس الأولوية بتوزيع زمن المعالج عليها بالتساوي.

يمكن إدخال مفهوم الأولوية الشُفعية لخوارزمية الشريط الدوار، وذلك بالسماح للإجراءات ذات الأولوية الأعلى بشُفع المعالج إذا كان ينفذ إجراءاً أقل أولوية لحظة وصولها. ويبين الشكل التالي مثالاً توضيحياً لهذه الفكرة:



الإجراءان **A** و **C** لهما نفس الأولوية والإجراء **B** له أولوية أعلى. يبدأ **A** بالتنفيذ لبعض الوقت، إلى أن يصل الإجراء **B** ذو الأولوية الأعلى ويقوم بشُفع **A** ويستمر بالتنفيذ حتى انتهائه. بعدها يعود **A** للتنفيذ إلى أن تنتهي حصته الزمنية وينتقل التنفيذ للإجراء **C** حتى انتهاء حصته الزمنية وهكذا ...

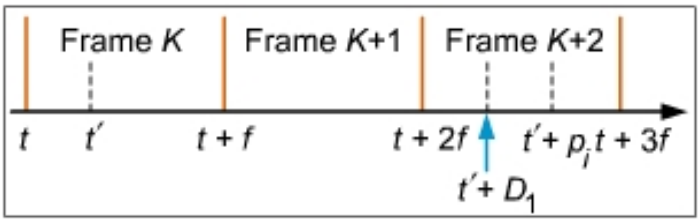
التنفيذي الدوري cyclic executive

يستعمل هذا الأسلوب أيضاً بكثرة في العديد من نظم الزمن الحقيقي لأنه بسيط ويولّد جدولةً قابلةً للتنبؤ. وهو يقوم بسلسلة ومراكبة تنفيذ إجراءات دورية حسب جدول محدد مسبقاً قبل التنفيذ. يمكن التعبير عن هذا الأسلوب ببساطة على أنه جدول استدعاءات لتوابع، كلّ منها هو إجراء، في حلقة غير منتهية.

يتخذ هذا الأسلوب قرارات الجدولة دورياً ومسبقاً، وليس في أية لحظة مثل الشريط الدوار. تسمى الفترات الزمنية (المتساوية) التي تفصل بين لحظات اتخاذ قرارات الجدولة بالأطر **frames** أو الأدوار الصغيرة **minor cycles**، ويرمز لطولها بالرمز *f*، ويسمى "حجم الإطار". الدور الكبير **major cycle or hyper period** هو المضاعف المشترك الأصغر لجميع أدوار إجراءات النظام.

بما أن قرارات الجدولة تؤخذ فقط في بداية كلّ إطار، لذلك لا يحدث أي شُفع داخل الإطار. إن طور **phase** أي إجراء دوري هو مضاعف صحيح غير سالب من حجم الإطار.

يبين الشكل التالي بعض القيود المفروضة على حجم الإطار *f*:



يجب أن يكون الإطار كبيراً كفاية بحيث يبدأ وينتهي أي إجراء ضمن إطار واحد. وهذا يقتضي أن يكون حجم الإطار f أكبر من كل أزمان التنفيذ e_i للإجراءات T_i. أي:

$$C_1 : f \geq \max_{1 \leq i \leq n}(e_i) \quad \textbf{(3.4)}$$

ولكي يكون طول الجدول الدوري أصغر ما يمكن، يجب أن يحتوي الدور الكبير على عدد صحيح من الأطر:

$$C_2 : \lfloor p_i / f \rfloor - p_i / f = 0 \quad \textbf{(3.5)}$$

وللتأكد من أن كل إجراء ينتهي قبل حده الزمني، يجب أن يكون الإطار صغيراً بحيث يكون هناك إطاراً واحداً على الأقل بين كل تحرير لإجراء وبين حده الزمني. العلاقة التالية مشتقة من الحالة الأسوأ التي تحدث عندما يبدأ دور إجراء ما مباشرة بعد بدء الإطار، ولا يمكن بالتالي

تحريره قبل بدء الإطار التالي:

$$C_3 : 2f - \gcd(p_i, f) \le D_i \quad \textbf{(3.6)}$$

حيث gcd هو القاسم المشترك الأعظم و D_i هو الحد الزمني النسبي للإجراء i.

سنوضح عملية حساب حجم الإطار f باستعمال مثالٍ عددي. ليكن لدينا نظاماً يحتوي على الإجراءات الدورية المبينة في الجدول التالي:

τ_i	p_i	e_i	D_i
τ_1	**15**	**1**	**14**
τ_2	**20**	**2**	**26**
τ_3	**22**	**3**	**22**

قيمة الدور الكبير لهذه الإجراءات هي 660 (وهو المضاعف المشترك الأصغر للقيم 15 و 20 و 22). نطبق الشروط الثلاثة السابقة C_1 و C_2 و C_3 فنحصل على العلاقات التالية:

$C_1 : \forall if \ge e_i \Rightarrow f \ge 3$

$C_2 : \lfloor p_i / f \rfloor - p_i / f = 0 \Rightarrow f = 2,3,4,5,10,...$

$C_3 : 2f - \gcd(p_i, f) \le D_i \Rightarrow f = 2,3,4,5$

نستنتج مما سبق أن القيم الممكنة لحجم الإطار ***f*** هي إما **3** أو **4** أو **5**.

جدولة الأولويات الثابتة – الخوارزمية "الرتيبة المعدل" Rate Monotonic (RM)

في النظم ذات الأولويات الثابتة، يعطى لكل إجراء أولوية محددة نسبةً لباقي الإجراءات. تُعتبر خوارزمية **RM** من أهم خوارزميات الجدولة في هذه النظم، وهي تعطي جدولةً أمثلية في نموذج الإجراءات المبسط المذكور سابقاً.

يعطى لكلّ إجراء في هذه الخوارزمية أولوية تتناسب عكساً مع دوره: فأولوية الإجراء الذي له دور كبير أصغر من أولوية الإجراء الذي له دور قصير.

سنشرح طريقة عمل هذه الخوارزمية على المثال التالي. ليكن لدينا مجموعة الإجراءات التالية:

τ_i	e_i	p_i	$u_i = e_i / p_i$
τ_1	**1**	**4**	**0.25**
τ_2	**2**	**5**	**0.4**
τ_3	**5**	**20**	**0.25**

نفترض أن زمن تحرير كل الإجراءات الثلاثة هو **0**. يبين الشكل التالي طريقة جدولة هذا النظام

باستعمال خوارزمية **RM**:

بما أن الإجراء τ_1 له أصغر دور، فإن أولويته أعلى من أولويات باقي الإجراءات، لذلك ينفذ أولاً. لاحظ أنه في اللحظة 4 يصل التكرار الثاني للإجراء τ_1 الذي يقوم بشُفع الإجراء τ_2 لأن أولويته أعلى.

تسمى القيمة u_i "الانشغالية" **utilization** وهي النسبة (من واحد) التي يشغِل فيها إجراء له دور p_i وزمن تنفيذ e_i المعالج. تذكر أن انشغالية المعالج في حالة n إجراء تعطى بالعلاقة **1.2** وهي:

$$u = \sum_{i=1}^{n} e_i / p_i \qquad (3.7)$$

بعض نتائج خوارزمية RM

من المهم عملياً معرفة ما هي الشروط الواجب تحققها في نظام أولويات ثابتة ليكون قابلاً للجدولة (أي لكي توجد جدولة واحدة على الأقل تحقق جميع القيود الزمنية لجميع إجراءات النظام).

تحدد النظرية التالية شروطاً على انشغالية النظام لكي يكون قابلاً للجدولة حسب خوارزمية **RM**: (نفترض أن الحد الزمني النسبي لأي إجراء يساوي دوره)

نظرية: تكون أية مجموعة من الإجراءات الدورية عددها n قابلة للجدولة حسب خوارزمية **RM** إذا كانت انشغالية المعالج **U** أصغر أو تساوي $n(2^{\frac{1}{2}}-1)$.

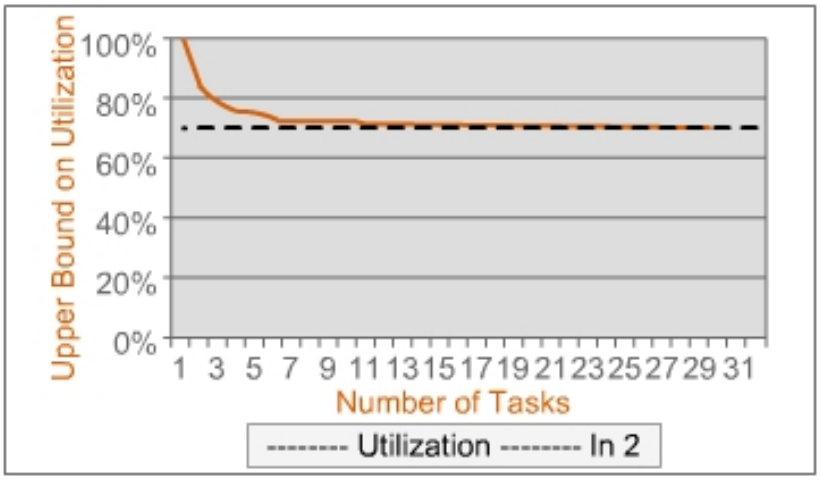
تجدر الملاحظة أنه عندما تصبح n كبيرة جداً، يمكن البرهان على أن قيمة الحد السابق تصبح:

$$\lim_{n\to\infty}(2^{\frac{1}{2}}-1) = \ln 2 \approx 0.69$$

يبين الجدول التالي قيم الحد السابق من أجل عدة قيم لـ n:

n	1	2	3	4	5	6	...	∞
الحد	**1.0**	**0.83**	**0.78**	**0.76**	**0.74**	**0.73**		**0.69**

يبين الشكل التالي مخططاً يبين كيفية تغير حد الانشغالية بدلالة عدد الإجراءات:



تجدر الإشارة إلى أن حد الانشغالية هو شرط كافٍ وغير لازم، أي أنه يمكن في بعض الأحيان بناء نظام له انشغالية أعلى من الحد ويكون قابلاً للجدولة حسب **RM** رغم ذلك.

على سبيل المثال، انشغالية النظام السابق المكون من الإجـراءات τ_1 وτ_2 وτ_3 تـساوي **0.9**، وهي أعلى من حد الانشغالية **0.78**، ورغم ذلك فهو قابل للجدولة حسب **RM** كما رأينا.

جدولة الأولويات الديناميكية – خوارزمية "الحد الأبكر أولاً" Earliest-Deadline-First (EDF)

على عكس نظم الأولويات الساكنة، تتغير أولويات الإجراءات أثناء عمل النظام في نظم الأولويات الديناميكية. تُعتبر خوارزمية **EDF** من أشهر خوارزميات هذه النظم، حيث تعتمد أولويات الإجراءات على حدودها الزمنية بدلاً من أزمان تنفيذها. في أية لحظة، الإجراء الذي له أقرب حد زمني هو الإجراء ذو الأولوية الأعلى.

تعطي النظرية التالية الشرط الواجب تحقيقه لكي تكون مجموعة إجراءات قابلة للجدولة حسب خوارزمية **EDF**.

نظرية: في نظام زمن حقيقي يحتوي على n إجراء دوري حدودها النسبية تساوي دورها، الشرط اللازم والكافي ليكون قابلاً للجدولة حسب خوارزمية **EDF** هو:

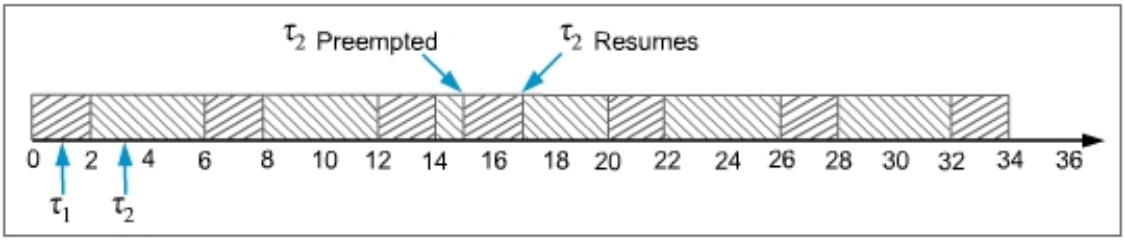
$$u = \sum_{i=1}^{n} \left(\frac{e_i}{p_i} \right) \leq 1 \qquad (3.8)$$

مثال:

لنأخذ على سبيل المثال نظاماً يحتوي على الإجرائين التاليين:

τ_i	p_i	e_i
τ_1	**2**	**5**
τ_2	**4**	**7**

يبين الشكل التالي طريقة جدولتها حسب خوارزمية **EDF**:



رغم أن لحظة تحرير كلا الإجرائين هي اللحظة **0**، يبدأ الإجراء τ_1 بالتنفيذ لأن له حد زمني أقرب. في اللحظة **t=2**، ينتهي تنفيذ τ_1 ويبدأ تنفيذ τ_2. في اللحظة **t=5**، يصل التكرار الثاني للإجراء τ_1، لكنه لا يقوم بشُفع τ_2 لأن حده الزمني أبعد من حد τ_2. في اللحظة **t=15**، يقوم τ_1 بشُفع τ_2 لأن حده الزمني (وهو **20**) أقرب من الحد الزمني للإجراء τ_2 (وهو **21**). يتابع τ_2 تنفيذه بعد انتهاء τ_1 في اللحظة **t=17**.

خصائص الخوارزمية EDF

إن خوارزمية **EDF** هي خوارزمية أمثلية في نظام شُفعي ذي معالج وحيد. أي أنه إذا وجِدت جدولة مقبولة للنظام، عندها ستعطي خوارزمية **EDF** جدولة مقبولة بالتاكيد.

مقارنة الخوارزميتين RM و EDF

تُقارن خوارزميات جدولة الإجراءات الدورية عادةً بمقدار انشغالية المعالج التي تنتج عنها. فمن المنطقي إبقاء المعالج مشغولاً قدر الإمكان طالما أن هناك عمل مفيد يقوم به بهدف تحقيق الحدود الزمنية للإجراءات. تتميز خوارزميات الأولوية الديناميكية بأنها أفضل من خوارزميات الأولوية الثابتة حسب هذا المعيار.

خوارزمية **EDF** أكثر مرونةً من **RM** وتعطي انشغاليةً أفضل. لكن عادةً ما تكون الجدولة الناتجة عن خوارزميات الأولوية الثابتة ذات تنبؤيةٍ أكبر من خوارزميات الأولوية الديناميكية. وفي حال زيادة العبء على النظام (التحميل الزائد)، تتميز **RM** بأنها أكثر ثباتاً من **EDF**، إذ أن

نفس الإجراء ذي الأولوية المنخفضة الذي يتعدى حده الزمني سيتعدى حده الزمني في المستقبل ولن تتأثر الإجراءات ذات الأولوية المرتفعة. بالمقابل، يصعب معرفة الإجراء الذي سيتعدى حده الزمني في نظام **EDF** عند حدوث التحميل الزائد.

تجدر الإشارة أيضاً إلى أن الإجراء الذي يتعدى حده الزمني في **EDF** تبقى أولويته أعلى من الإجراءات التي لم تتعدى حدها الزمني بعد. وإذا استمر هذا الإجراء بالتنفيذ، يمكن أن يؤدي ذلك إلى تعدي الإجراءات الأخرى لحدودها الزمنية. من الأفضل في هذه الحالة أن يترك هذا الإجراء المعالج أملاً بتلبية الحدود الزمنية للإجراءات الأخرى.

على سبيل المثال، يبين الجدول التالي مجموعة إجراءات دورية في نظام زمن حقيقي. عند جدولة هذه الإجراءات حسب خوارزمية **EDF**، سيتعدى كل من الإجرائين τ_1 و τ_2 حدودهما الزمنية إذا سمحنا للإجراء الذي تعدى حده الزمني بالاستمرار بالتنفيذ. بينما سيتعدى τ_2 فقط حده الزمني إذا استعملنا إجرائية **RM**.

τ_i	r_i	e_i	p_i
τ_1	**0**	**2**	**5**
τ_2	**0**	**4**	**6**

3. مزامنة الإجراءات ومشاركة المصادر

افترضنا حتى الآن أن إجراءات النظام مستقلة ويمكن أن يجري شُغلها في أية لحظة. هذه الفرضيات غير واقعية عملياً في معظم النظم، إذ تحتاج الإجراءات أن تتصل وتتفاعل مع بعضها البعض.

سنقدم في هذا القسم المشاكل الناتجة عن مشاركة المصادر وخوارزميات الجدولة المستعملة في هذه الحالة.

المصادر Resources

المصادر هي أية تسهيلات يقدمها النظام للإجراءات لمساعدتها في تأدية عملها. وفيما يلي بعض الأمثلة على المصادر المشتركة في النظام:

- الملفات.
- التجهيزات المختلفة (طابعات، شاشات، ...).

- المتحولات المشتركة في الذاكرة التي غالباً ما تُستعمل للاتصال بين الإجراءات.

هناك الكثير من التقنيات البرمجية المستعملة كمتحولات مشتركة في الذاكرة لتسهيل الاتصال بين الإجراءات نذكر منها:

- الدارئ **buffer** بأنواعه المتعددة (الحلقي **circular**، المضاعف **double**، ...).
- صندوق البريد **mail box**.
- الأرتال **queues**.

الأقسام الحرجة critical sections

مهما كان المصدر الذي يحتاج الإجراء للوصول إليه، عادةً ما يحتاج أن يصل إليه وحيداً ودون مقاطعة، وإلا يمكن أن تحدث حالات خاصة تجعل نتيجة التنفيذ غير صحيحة. سنبين في المثال التالي كيف يمكن أن يؤدي الوصول إلى مصدر مشترك من قِبل إجرائين في نفس الوقـت دون تنسيق (مزامنة) إلى نتائج خاطئة. لنفترض أنه لدينا الإجرائين التاليين τ_1 و τ_2:

τ_1	τ_2
(1) x = i; **(2) x = x + 1;** **(3) i = x;**	**(4) y = i;** **(5) y = y - 1;** **(6) i = y;**

المتحول **x** محلي في الإجراء τ_1 والمتحول **y** محلي في الإجراء τ_2. المتحول **i** هـو متحـول مشترك بين الإجرائين قيمته الابتدائية صفر وهو المصدر المشترك الذي نتحدث عنه.

إذا فرضنا أن كلاً من الإجرائين ينفذ مرة واحدة، تكون القيمة النهائية الـصحيحة فـي **i** هـي الصفر.

لكن لو فرضنا أن τ_1 بدأ بالتنفيذ وجرى شُفعه بين التعليمتين **(1)** و **(2)**، وانتقل التنفيذ للإجراء τ_2، ثم عاد التنفيذ للإجراء τ_1 بعد الانتهاء من التنفيذ الكامل للإجراء τ_2، عندها تكون القيمة النهائية في **i** تساوي **1**، وهي قيمة خاطئة.

تسمى مجموعة التعليمات التي يتعامل الإجراء فيها مع المصدر المشترك "القسم الحرج" **critical section**.

يمكن أن يحتوي إجراء ما على أقسام حرجة لنفس المصدر أو لمصادر مختلفة. وإذا دخل إجراءان قسمين حرجين لنفس المصدر وفي نفس الوقت، فإن ذلك يؤدي لحدوث أخطاء في نتائج التنفيذ، مما يمكن أن يؤدي لنتائج كارثية في بعض الحالات.

لذلك، هناك حاجة كبيرة لتقنيات تمنع حدوث مثل هذا الوضع في معظم نظم الزمن الحقيقي العملية. ومن هذه التقنيات نذكر:

- "العلاّم" **semaphore**.
- "المراقب" **monitor**.
- تعليمة "اختبر وضع" **test and set**.

سنناقش تقنية العلاّم بالتفصيل لبساطتها وفعاليتها ولكونها موجودة في معظم نظم التشغيل.

العلاّمات semaphores

العلاّم **S** هو موقع في الذاكرة يعمل كقفل يحمي القسم الحرج المتعلق بمصدر ما. وعادةً ما تكون قيمة **S** ثنائية **binary** (صفراً أو واحداً):

- عندما تكون **S=1**، يدل ذلك على أن القفل مفتوح ويمكن لأي إجراء الدخول للقسم الحرج.
- وعندما **S=0**، يدل ذلك على أن القفل مغلق ويوجد حالياً إجراء واحد يعمل في القسم الحرج، ويجب على جميع الإجراءات التي تريد الدخول إليه الانتظار ريثما يخرج ذلك الإجراء منه.

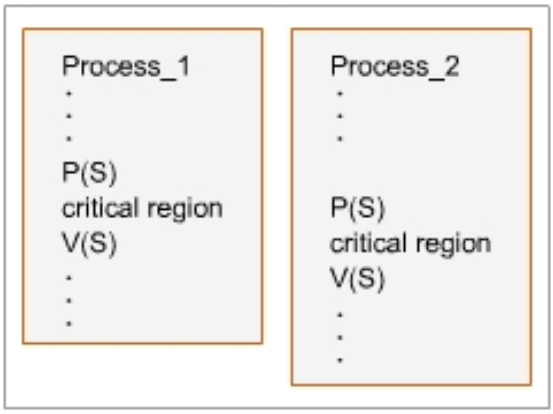
يعرف على العلاّم عمليتان أساسيتان هما **P(S)** (إنتظار وإغلاق) و **V(S)** (تحرير). وفيما يلي خوارزمية عمل كل منهما:

```
void P(int S)
{
    While (S==0);
    S=0;
}
void V(int S)
{
    S=1;
}
```

تفحص العملية **P(S)** قيمة العلاّم **S**. طالما بقيت قيمته تساوي **0**، يبقى الإجراء يدور في حلقة **while** بانتظار أن تصبح قيمة **S** تساوي **1** (أي يجمد الإجراء الذي استدعى **P(S)** في هذه الحالة). عندما تصبح **S=1**، يخرج الإجراء من حلقة الانتظار ويقفل العلاّم (بجعل قيمته تساوي **0**). تقوم العملية **V(S)** بفتح العلاّم بجعل قيمته تساوي **1**.

تُستعمل العمليتان **P(S)** و **V(S)** لتحقيق التنسيق بين الإجراءات كما يلي: قبل الدخول إلى القسم الحرج، يستدعي الإجراء العملية **P(S)** وعند الانتهاء منه، يستدعي **V(S)** كما هو مبين في الشكل

التالي:



تجدر الإشارة إلى أننا نحتاج لعلاّم مستقل لحماية كلّ مصدرٍ مشترك. وإذا احتاج قسمٌ حرجٌ ما للوصول إلى عدة مصادر مشتركة، فعندها يجب استدعاء العملية **P** من أجل كلّ العلاّمات الموافقة لهذه المصادر قبل الدخول إليه لحجزها كلها.

غالباً ما تكون العمليتان **P** و **V** جزءاً من نواة نظام التشغيل أو استدعاءات لخدمات النظام، لأنها يجب أن تُنفذ دون مقاطعة لكي تعمل بشكل صحيح. وسنبين أهمية هذا الأمر في المثال التالي.

لنفترض أن التابع **P** السابق قد جرى ترجمته إلى لغة الآلة كما يلي:

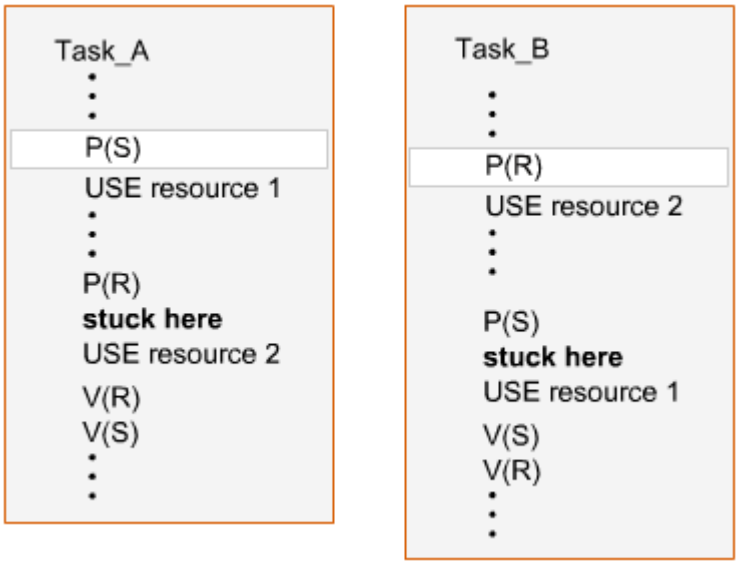
```
@1      LOAD  R1,S
        TEST  R1,0
        JEQ   @1
        STORE S,0
```

نفترض كذلك أنه يوجد إجراءان يريدان الدخول للقسم الحرج ويستدعيان بالتالي التابع **P** قبل الدخول. القيمة الابتدائية للعلاّم **S** تساوي **1**، مما يعني أن القسم الحرج حر. يبدأ الإجراء الأول بتنفيذ تعليمات لغة الآلة السابقة. قبل أن يصل التنفيذ لتعليمة **STORE**، تحدث مقاطعة وينتقل التنفيذ للإجراء الثاني الذي يستدعي نفس التابع السابق. بما أن الإجراء الأول قوطع قبل تخزين القيمة **0** في **S**، يجد الإجراء الثاني أن قيمة **S** مازالت **1**، فيجعلها **0** ويدخل في القسم الحرج. قبل أن يفرغ هذا الإجراء من استعمال المصدر المشترك والخروج من القسم الحرج، تحدث مقاطعة ويعود التنفيذ للإجراء الأول الذي يتابع التنفيذ من تعليمة **STORE** التي توقف عندها ويدخل في القسم الحرج. بهذا يدخل كلا الإجرائين إلى القسم الحرج مما يمكن أن يؤدي لمشاكل جدية كما ذكرنا سابقاً.

أخيراً، على الرغم من بساطة العلامات، يجب استعمالها بحذر وفهم عملها جيداً، وإلا أدى سوء استعمالها لحدوث مشاكل منطقية مثل "المنع المتبادل" **deadlock** الذي سنتحدث عنه فيما يلي.

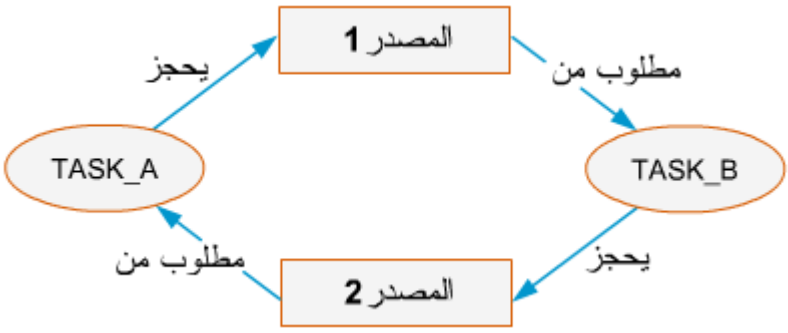
المنع المتبادل Deadlock

يمكن أن تحدث هذه المشكلة الخطيرة عندما تحاول عدة إجراءات الوصول إلى مصدرين على الأقل، مما يؤدي لتوقفها كلها إذا جرى تنفيذها بحالةٍ خاصة سنوضحها في المثال التالي.
لنفترض أنه لدينا الإجرائين التاليين:



يحتاج الإجراءان **TASK_A** و **TASK_B** الوصول إلى مصدرين يحميهما العلاّمان **R** و **S**.
يحجز الإجراء **TASK_A** المصدر الأول بعد استدعاء **P(S)** ويبدأ باستعماله. وقبل أن يصل إلى الاستدعاء **P(R)** تحدث مقاطعة وينتقل التنفيذ إلى **TASK_B** الذي يحجز المصدر الثاني ويستعمله. عندما يصل التنفيذ إلى **P(S)**، يتوقف **TASK_B** عن التنفيذ لأن الإجراء **TASK_A** يحجز المصدر الأول. عندها ينتقل التنفيذ إلى الإجراء **TASK_A** الذي يصل بدوره للاستدعاء **P(R)** ويتوقف لأن **TASK_B** يحجز المصدر الثاني. بهذا يحجز كل إجراء مصدر وينتظر المصدر الذي يحجزه الإجراء الآخر إلى ما لا نهاية.

يمكن التعبير عن هذه الحالة بالمخطط التالي:



إن المنع المتبادل مشكلة خطيرة جداً لأنها تحدث باحتمالٍ صغيرٍ جداً ومن الصعب معرفة أنها موجودة بأي اختبارٍ إلى أن تحدث.

كذلك فإن حلولها ليست سهلةً ولها نتائج عديدة غير مرغوبة، من أهمها إضافة عبءٍ حسابي على النظام يؤدي لإضعاف أدائه.

سوف نتطرق إلى إحدى الطرق الشهيرة لتفادي حدوث المنع المتبادل والمسماة "بخوارزمية المصرفي" **banker's algorithm**. ويمكن الاطلاع على حلول المنع المتبادل الأخرى في الكتب التي تدرس نظم التشغيل بشكل عام.

خوارزمية المصرفي the banker's algorithm

تُستعمل هذه الخوارزمية لتفادي الحالات الغير آمنة التي يمكن أن تؤدي لحدوث المنع المتبادل.

اقترحها ديجكسترا **Dijkstra** عام **1968** وهي مشابهة لعمل المصارف في المدن الصغيرة من حيث الحفاظ على مخزون من السيولة النقدية على سبيل الاحتياط.

يودع الناس أموالهم في المصرف ويمكنهم سحبها في أية لحظة. لا يحتفظ المصرف بكل الأموال السائلة في حوزته، بل يستثمر **95%** منها ويحتفظ بنسبة **5%** كسيولة نقدية احتياط في حال أراد بعض المودعين سحب جزء من إيداعاتهم.

إذا طلب العديد من الزبائن سحب إيداعاتهم، عندها لن يتمكن المصرف من تلبية جميع الطلبات.

تفترض خوارزمية المصرفي وجود عدة مصادر من نوع واحد فقط، لكن يمكن تمديدها بحيث تعمل مع عدة أنواع من مصادر.

تتأكد الخوارزمية في كل لحظة من أن عدد المصادر المحجوزة من قِبل إجراءات النظام زائد عدد المصادر التي يحتاج إليها أحد هذه الإجراءات لإتمام تنفيذه لا يتجاوز العدد الكلي المتاح من المصدر. نقول في هذه الحالة أن النظام في حالة آمنة.

وبالعكس، يكون النظام في حالة غير آمنة إذا كان المجموع السابق أكبر تماماً من العدد الكلي المتاح من المصدر.

من الممكن لكن ليس من الضروري أن تؤدي الحالات الغير آمنة إلى منع متبادل، لكن من المؤكد أن الحالات الآمنة لن تؤدي أبداً لحدوث منع متبادل. لذلك تضمن الخوارزمية بقاء النظام في حالة آمنة وتضمن بالتالي عدم حدوث المنع المتبادل.

لنأخذ على سبيل المثال نظاماً مكوناً من ثلاثة إجراءات **A** و **B** و **C** و **10** مصادر من نوع واحد (كتل ذاكرة مثلاً). إذا علمنا أن الإجراء **A** لن يحتاج لحجز أكثر من **6** مصادر في نفس الوقت ولن يحتاج **B** لأكثر من **5** ولن يحتاج **A** بأكثر من **7**. يحتاج النظام لبناء جدول مشابه للجدول التالي في كل لحظة لتتبع المصادر المتاحة والمطلوبة في تلك اللحظة.

الإجراء	العدد الأعظمي المطلوب	عدد المصادر المحجوزة حالياً	العدد الذي من الممكن أن يحتاج إليه
A	6	0	6
B	5	0	5
C	7	0	7
العدد الكلي المتاح حالياً			10

عند كل طلب حجز للمصادر، يحدث نظام التشغيل هذا الجدول للتأكد من أن النظام ما زال في وضع آمن. فيما يلي مثال على وضع آمن للنظام:

الإجراء	العدد الأعظمي المطلوب	عدد المصادر المحجوزة حالياً	العدد الذي من الممكن أن يحتاج إليه
A	6	2	4
B	5	3	2
C	7	1	6
العدد الكلي المتاح حالياً			4

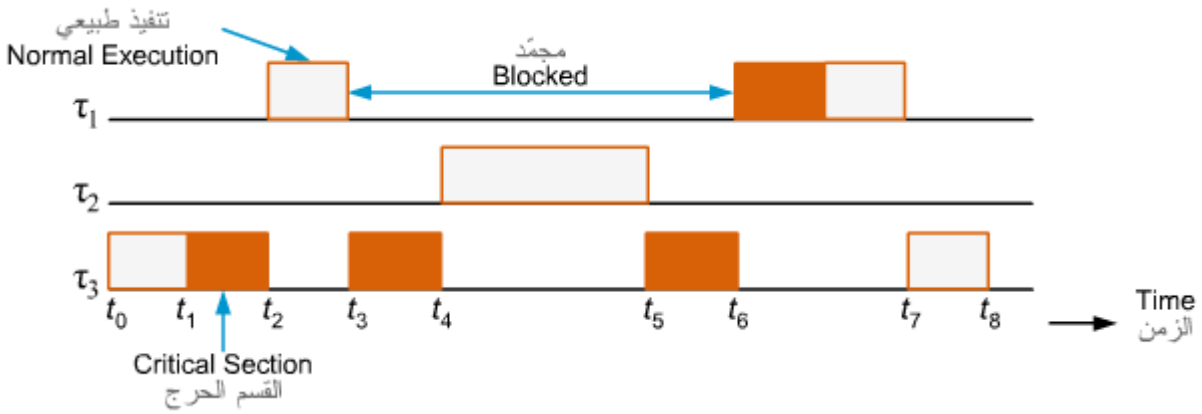
هذه الحالة آمنة لأن العدد الكلي المتاح حالياً (وهو **4**) أكبر أو يساوي العدد الذي من الممكن أن يحتاج إليه أحد الإجراءات لإتمام تنفيذه (مثل الإجراء **B** على سبيل المثال، وكذلك الإجراء **A**). فيما يلي مثال على حالة غير آمنة لأن العدد الكلي المتاح لا يكفي أي إجراء، ومن الممكن إذاً أن يحدث منع متبادل:

الإجراء	العدد الأعظمي المطلوب	عدد المصادر المحجوزة حالياً	العدد الذي من الممكن أن يحتاج إليه
A	6	4	2
B	5	3	2
C	7	2	5
العدد الكلي المتاح حالياً			1

انعكاس الأولوية priority inversion

من أهم نتائج استعمال تقنيات مزامنة الإجراءات كالعلاّمات وغيرها في نظم الزمن الحقيقي حدوث انعكاس لأولوية إجراءات الزمن الحقيقي، حيث ينفذ الإجراء ذو الأولوية المنخفضة عوضاً عن الإجراء ذي الأولوية المرتفعة فقط لأنه يحجز مصدراً يحتاج إليه ذلك الإجراء. هذا الأمر غير مرغوب فيه نهائياً لأنه يمكن أن يؤدي لأن يتعدى إجراء حساس جداً حده الزمني بسبب إجراء ذي أولوية منخفضة. ونبين هذه الحالة في المثال التالي:

لدينا نظام يحتوي على ثلاثة إجراءات τ_1 و τ_2 و τ_3. أولوية τ_1 أعلى من أولوية τ_2 وأولوية τ_2 أعلى ممن أولوية τ_3. يتشارك τ_1 و τ_3 بمصدر مشترك ويحتاجان لتقنية مزامنة تنظم وصولهما إليه (علاّم مثلاً مع عمليات **P** و **V** الخاصة به). لا يحتاج τ_2 للوصول إلى هذا المصدر. لنفترض أنه حدث تنفيذ الإجراءات حسب المخطط الزمني التالي (القسم الحرج ملون باللون الغامق).



يبدأ τ_3 العمل في اللحظة t_0 ويقوم بقفل العلاّم في اللحظة t_1 لأنه يريد العمل مع المصدر المشترك. يصل الإجراء τ_1 في اللحظة t_2 ويقوم بشُفع الإجراء τ_3 قبل أن ينتهي من القسم الحرج لأن له أولوية أعلى. في اللحظة t_3، يحاول الإجراء τ_1 قفل العلاّم لأنه يريد أيضاً العمل مع المصدر المشترك، لكنه يدخل في حالة انتظار (يصبح مجمداً) لأن العلاّم محجوز مسبقاً من قِبل τ_3. عندها، يعود التنفيذ إلى τ_3 ليتابع قسمه الحرج. لكن في اللحظة t_4 يصل الإجراء τ_2 الذي له أولوية أعلى من τ_3، فيقوم بشُفعه. وبما أن τ_2 لا يحتاج للوصول إلى المصدر المشترك، يستمر تنفيذه إلى أن ينتهي في اللحظة t_5، مطيلاً بذلك زمن انتظار τ_1 ذي الأولوية الأعلى بمقدار زمن تنفيذ τ_2. يعود بعدها التنفيذ للإجراء τ_3 إلى أن ينتهي القسم الحرج في اللحظة t_6 ويحرر العلاّم. عندها ينتقل التنفيذ للإجراء τ_1 لأن أولويته أعلى ولأن العلاّم الذي ينتظره أصبح حراً. بعد أن ينتهي تنفيذ الإجراء τ_1 في اللحظة t_7، يعود التنفيذ للإجراء τ_3 حتى

ينتهي تنفيذه في اللحظة t_8.

تصور لو أنه كان هناك عدد أكبر من الإجراءات المشابهة للإجراء τ_2 بأن لها أولوية أعلى من τ_3 وأقل من τ_1. من الممكن في هذه الحالة أن تزيد هذه الإجراءات من زمن انتظار τ_1 لفترة طويلة رغم أولويته المرتفعة.

نقول بشكل عام أنه حصلت حالة إنعكاس أولوية خلال فترة زمنية ما إذا مِنع إجراء ذو أولوية مرتفعة من التنفيذ من قِبل إجراءات لها أولوية أقل. حصل هذا الأمر في المثال السابق في المجال الزمني $[t_3,t_6]$ حين مِنع τ_1 من التنفيذ من قِبل τ_2 و τ_3.

من الواضح أنه من غير المرغوب به أن يحصل انعكاس للأولوية في نظم الزمن الحقيقي. لذلك، تُستعمل في هذه الحالة خوارزميات جدولة مختلفة عن الخوارزميات التي رأيناها سابقاً بهدف التقليل قدر الإمكان من هذه الظاهرة. سنناقش فيما يلي خوارزميتين شهيرتين في هذا المجال هما "وراثة الأولوية" و "سقف الأولوية".

خوارزمية وراثة الأولوية priority inheritance

تتغير أولويات الإجراءات في هذه الخوارزمية ديناميكياً بحيث يأخذ الإجراء الذي يعمل ضمن القسم الحرج الأولوية العظمى للإجراءات التي تنتظر خروجه من القسم الحرج.

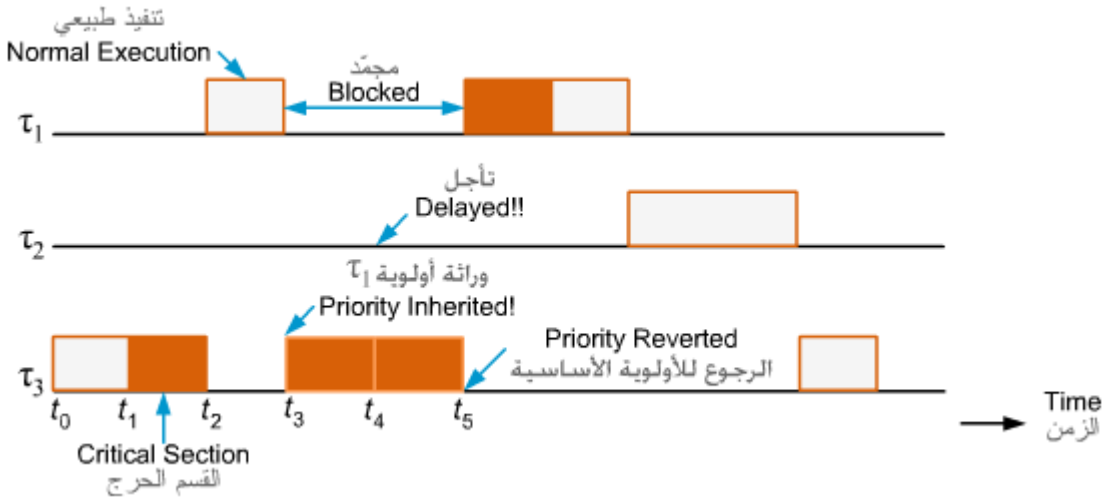
بكلمات أخرى، عندما يجمد إجراء ما τ_i إجراءات أخرى ذات أولوية أعلى، يرث مؤقتاً أعلى أولوية من أولويات هذه الإجراءات إلى أن يخرج من القسم الحرج حيث يستعيد أولويته الأصلية.

الخطوات الأساسية للخوارزمية:

- سيدخل أي إجراء مهما كانت أولويته في حالة انتظار إذا حاول حجز علاّم محجوز من قِبل إجراء آخر.
- إذا جمد إجراء τ_1 من قِبل إجراء آخر τ_2، وكانت أولوية τ_1 أعلى من أولوية τ_2، يرث τ_2 أولوية τ_1 طالما بقي مجمداً له، ويعمل وفق هذه الأولوية الجديدة الموروثة. عندما يخرج τ_2 من القسم الحرج الذي سبب هذا التجميد، تعود أولويته إلى القيمة التي كانت عليها قبل دخول القسم الحرج.
- وراثة الأولوية متعدية. أي إذا كان τ_2 يجمد τ_2 و τ_2 يجمد τ_1، وأولوية τ_2 أقل من أولوية τ_2 وأولوية τ_2 أقل من أولوية τ_1، عندها يرث τ_3 أولوية τ_1 عبر τ_2.

ففي المثال السابق، إذا استعملنا فكرة وراثة الأولوية، ستزيد أولوية τ_3 لتصبح مساويةً لأولوية τ_1 في اللحظة t_3 التي يجمد فيها الإجراء τ_1. الإجراء τ_1 وعندها، لن يتمكن τ_2 من شُنع τ_3 عندما يصل لأن أولوية τ_3 أصبحت أعلى من أولوية τ_2.

يبين الشكل التالي المخطط الزمني الناتج عن تطبيق فكرة وراثة الأولوية على المثال السابق:



تجدر الإشارة أخيراً إلى أن خوارزمية وراثة الأولوية لا تمنع حدوث المنع المتبادل على عكس خوارزمية "سقف الأولوية" التي سنناقشها في الفقرة التالية.

خوارزمية سقف الأولوية priority ceiling

هذه الخوارزمية هي تحسين لخوارزمية وراثة الأولوية، بحيث يمنَع أي إجراء من الدخول إلى القسم الحرج إذا كان هذا سيؤدي إلى تجميده. يعطى لكل مصدر أولوية (تسمى سقف الأولوية) تساوي أعلى أولوية لإجراء سيستعمل هذا المصدر.

خطوات خوارزمية سقف الأولوية هي نفسها خطوات خوارزمية وراثة الأولوية، مع فرق واحد هو أنه يمكن لإجراء τ_i أن يمنَع من دخول القسم الحرج إذا وجِد علاّم محجوز من قِبل إجراء آخر وله سقف أولوية أكبر أو يساوي أولوية τ_i.

وسنبين هذا الفرق في المثال التالي:

ليكن لدينا نظام مؤلف من ثلاثة إجراءات τ_1 و τ_2 و τ_3 ويوجد فيه المصادر التالية:

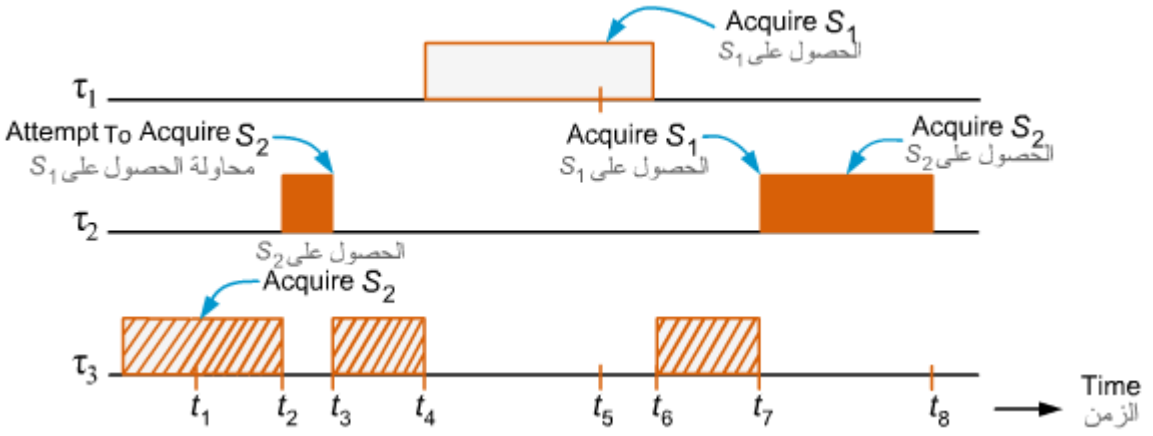
القسم الحرج	الإجراءات التي تصل إليه	سقف الأولوية
S_1	τ_1, τ_2	$p(\tau_1)$
S_2	τ_1, τ_2, τ_3	$p(\tau_1)$
S_3	τ_3	$p(\tau_3)$
S_4	τ_2, τ_3	$p(\tau_2)$

إذا فرضنا أن τ_2 يحجز حالياً المصدر S_2، ويبدأ الإجراء τ_1 بالعمل وطلب حجز المصدر S_1، عندها سيجمد τ_1 ولن يحجز المصدر S_1 رغم أنه حر وذلك لأن أولويته ليست أكبر تماماً من سقف أولوية المصدر المحجوز S_2.

لنأخذ كمثالٍ آخر نظاماً فيه ثلاثة إجراءات أيضاً هي τ_1 و τ_2 و τ_3. أولوية τ_1 أكبر من أولوية τ_2 وأولوية τ_2 أكبر من أولوية τ_3. تقوم هذه الإجراءات بالعمليات التالية:

- τ_1: حجز S_1، تحرير S_1
- τ_2: حجز S_1، حجز S_2، تحرير S_2، تحرير S_1
- τ_3: حجز S_2، تحرير S_2

حسب قاعدة تحديد قيمة سقف الأولوية، تكون قيمة سقف أولوية S_1 تساوي $p(\tau_1)$ و قيمة سقف أولوية S_2 تساوي $p(\tau_2)$. ويبين الشكل التالي جدولة هذه الإجراءات حسب خوارزمية سقف الأولوية:



نفترض أن τ_3 يبدأ التنفيذ أولاً ويحجز المصدر S_2 في اللحظة t_1 ويدخل في القسم الحرج.

يبدأ τ_2 التنفيذ في اللحظة t_2 ويقوم بشُفع τ_3 لأن أولويته أعلى.

يحاول بعد ذلك حجز المصدر S_1 في اللحظة t_3. في هذه اللحظة، يجمد τ_2 لأن أولويته ليست أعلى من سقف أولوية S_2 الذي يحجزه الإجراء τ_3، ويرث τ_3 أولوية τ_2 مؤقتاً ويتابع تنفيذه.

في اللحظة t_4، يصل الإجراء τ_1 ويقوم بشُفع τ_3 ويستمر بالتنفيذ حتى اللحظة t_5 حيث يحاول حجز المصدر S_1.

يعطى المصدر S_1 للإجراء τ_1 لأن أولويته أعلى من أسقف أولويات جميع المصادر المحجوزة في هذه اللحظة (وهي فقط المصدر S_2 الدي يحجزه τ_3).

ينتهي تنفيذ الإجراء τ_1 في اللحظة t_6 ويعود التنفيذ إلى τ_3 الذي يتابع تنفيذه حتى ينتهي في اللحظة t_7.

عندها يسمح للإجراء τ_2 بحجز S_1 ثم S_2 ومتابعة التنفيذ حتى ينتهي في اللحظة t_8.

نتائج خوارزميات التحكم بالوصول للمصادر

يمكن لإجراء أن يجمد من قِبل إجراءٍ ذي أولويةٍ أقلّ لمرةٍ واحدةٍ فقط ولفترةٍ لا تزيد عن طول قسمٍ حرجٍ واحد.

والشرط التالي كافٍ لاختبار إمكانية جدولة **n** إجراءٍ دوري حسب خوارزمية **RM**:

$$\sum_{i=1}^{n}\left(\frac{e_i}{p_i}\right)+\max\left(\frac{B_1}{p_1},...,\frac{B_{n-1}}{p_{n-1}}\right)\le n(2^{\frac{1}{n}}-1)$$

حيث B_i هو زمن التجميد الأعظمي الذي يمكن للإجراء τ_i أن يتعرض له من قِبل إجـ ـراءٍ ذي أولويةٍ منخفضة، و p_i هو دور الإجراء τ_i.

مقرر: نظم الزمن الحقيقي

الفصل الثالث: تصميم برمجيات نظم الزمن الحقيقي

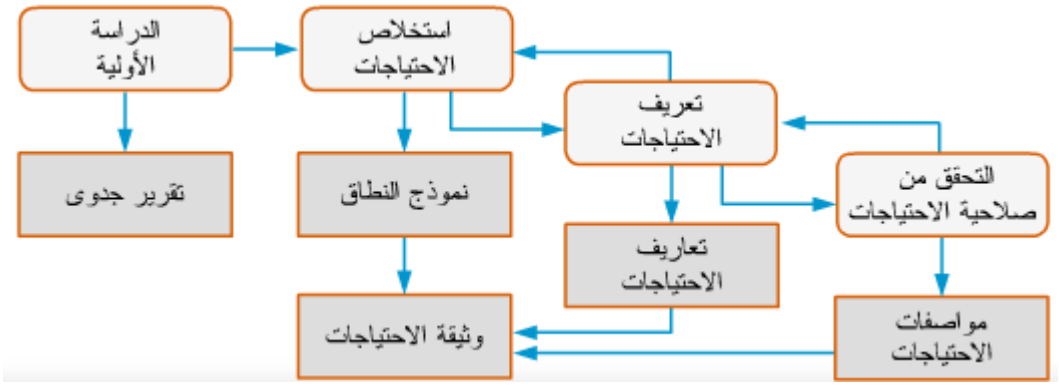
فهرس الفصل:

1. هندسة الاحتياجات

إن هندسة الاحتياجات هي جزء من علم هندسة البرمجيات. وهو يهتم بتحديد أهداف ومهام والقيود المفروضة على النظم البرمجية، وتمثيل هذه المعلومات بأشكال مناسبة للنمذجة والتحليل.

هدفها الأساسي إذاً هو إنشاء توصيفٍ كاملٍ وصحيحٍ ومفهومٍ لكل من الزبون والمطور لاحتياجات النظام البرمجي. وهذا الأمر يحتوي على تناقض بشكل من الأشكال، إذ يجب على هندسة الاحتياجات تقديم توصيف سهل الفهم للزبون لكي يتأكد من أن النظام قيد التطوير يلائم تماماً احتياجاته وتوقعاته، ويجب كذلك أن تحتوي على توصيف كامل للمهام والقيود المفروضة على النظام لتكون أساساً يعتمد عليه المطورون. وفي حالة نظم الزمن الحقيقي، يضاف إلى كل ما سبق التعقيدات الناتجة عن الحاجة لتمثيل قيود الأداء والزمن.

يبين الشكل التالي المخطط العام لتدفق العمل في مرحلة هندسة الاحتياجات من مراحل بناء نظام الزمن الحقيقي:



تُمثل الفعاليات على شكل مستطيلات ذات حواف ناعمة، بينما تُمثل الوثائق الناتجة عن هذه الفعاليات على شكل مستطيلات رمادية.

يبدأ إجراء هندسة الاحتياجات بمرحلة الدراسة الأولية، وتتضمن القيام ببحث حول دوافع المشروع وطبيعة المشكلة والقيود المفروضة عليها وتحديد نطاق العمل والأولويات المستقبلية، بالإضافة إلى تحليلٍ مبكر للقيود الزمنية المفروضة على النظام إذا كان النظام يعمل في الزمن الحقيقي. الوثيقة الأساسية الناتجة عن هذه المرحلة هو تقرير الجدوى الذي من الممكن أن يقترح عدم جدوى المتابعة في تطوير هذا النظام، وهو أمر لا يحدث في غالبية الأحيان.

ينتقل الإجراء بعدها لمرحلة استخلاص الاحتياجات، وهي تتضمن عمليات جمع الاحتياجات والمتطلبات باستعمال أساليب متعددة مثل مقابلة الزبون وطرح الأسئلة المفصلة عليه وإجراء النماذج الأولية للمشروع **prototypes**. يمكن التعبير عن الاحتياجات المستخلصة من هذه المرحلة بأساليب متعددة تتراوح بين النص العادي المكتوب بلغة إنسانية والتوصيف الصوري الرياضي المجرد. لكن، عادةً ما يعبر عنها باستعمال نموذج النطاق، وهو نموذج يتضمن مخططات عديدة مثل مخطط حالات الاستعمال **use-case**، والمخططات العلاقاتية ومخططات السياق **context**.

المرحلة التالية هي مرحلة تعريف الاحتياجات. فمن الضروري تعريف كل الاحتياجات المستخلصة تعريفاً دقيقاً بحيث يمكن تحليلها لاحقاً للتحقق من صلاحيتها وكفايتها وعدم تناقضها. خرج هذه المرحلة هي وثيقة متطلبات تحتوي على مواصفات الحاجات البرمجية **Software Requirement Specification (SRS)** التي تصف مزايا وتصرف النظام النهائي، بالإضافة إلى القيود المفروضة عليه.

2. أنواع الاحتياجات

على الرغم من وجود العديد من التصنيفات للاحتياجات، فإن التصنيف المتبع عـادة فـي نظـام الزمن الحقيقي هو التصنيف **IEEE830** المعرف من قبل المنظمة **IEEE**. يعرف هذا التـصنيف الأنواع التالية من الاحتياجات[1]:

1. **احتياجات وظيفية functional**:
2. **الواجهات الخارجية**:
3. **الأداء**:
4. **قاعدة المعطيات المنطقية**:
5. **قيود التصميم**:
   - مدى التوافق مع المعايير.
   - **واصفات النظم البرمجية:**
     - الوثوقية **Reliability**.

[1] تسمى الاحتياجات من 2 إلى 6 بالاحتياجات الغير وظيفية.

- الإتاحية **Availability**.
- الأمن **Security**.
- قابلية الصيانة **Maintainability**.
- الناقلية **Portability**.

تشمل الاحتياجات الوظيفية توصيفاً لجميع مداخل النظام، بالإضافة إلى سلسلة العمليات الواجب تنفيذها والخرج الموافق لكل قيمة دخل ممكنة، سواء كانت هذه القيمة طبيعية أو غير طبيعيـة. يمكن أن تشمل الحالات الغير طبيعية توصيفاً لكيفية معالجة الأخطاء ولاسـيما حـالات تعـدي إجراءات الزمن الحقيقي لحدودها الزمنية.

بكلمات أخرى، تصف الاحتياجات الوظيفية التصرف الحتمي الكامل للنظام.

توصف احتياجات الواجهات الخارجية جميع مداخل ومخارج النظام، وتشمل:

- الاسم.
- الهدف.
- مصدر الدخل أو وجهة الخرج.
- المجالات المقبولة والدقة والسماحية.
- وحدة القياس.
- التوقيت.
- العلاقات مع المداخل والمخارج الأخرى.
- أنماط المعطيات.
- أنماط الأوامر.

تشمل احتياجات الأداء الاحتياجات الديناميكية والساكنة المفروضة علـى النظـام البرمجـي أو تفاعل المستخدم مع النظام البرمجي ككل.

يمكن أن تشمل الاحتياجات الساكنة في نظم الزمن الحقيقي عدد المـستخدمين الـذين يمكـنهم استخدام النظام في نفس الوقت، بينما يمكن أن تشمل الاحتياجات الديناميكيـة عـدد المنـاقلات **transactions** وحجم المعطيات التي يمكن معالجتها في فترة زمنية محددة في الحالات العاديـة وحالات العبء الزائد المحمل على النظام.
تتضمن احتياجات قواعد المعطيات المنطقية تعريف أنماط المعطيات المستعملة فـي الوظـائف

المختلفة للنظام ومعدل استعمالها وسماحيات الوصول وكائنات المعطيات **entity** والعلاقات بينها وشروط تكامل المعطيات.

تتعلق احتياجات قيود التصميم بمدى المطابقة مع المعايير وبحدود البينان المادي المستعمل، بينما تتعلق احتياجات واصفات النظم البرمجية بالوثوقية والإتاحية والأمن وقابلية الصيانة والناقلية.

3. توصيف احتياجات نظم الزمن الحقيقي

يبدو أنه لا يوجد أسلوب موحد لتوصيف احتياجات نظم الزمن الحقيقي. ويستعمل مهندسو نظم الزمن الحقيقي عادةً أحد الطرق التالية أو خليطاً منها:

- التحليل من الأعلى للأسفل **top-down** أو ما يسمى بالتحليل المهيكل **structured analysis**.
- الأساليب الغرضية التوجه.
- لغات توصيف البرامج أو شبه البرنامج **pseudo-code**.
- التوصيفات الوظيفية عالية المستوى.
- أساليب أخرى عملية مثل اللغات المحكية والتوصيف الرياضي.

يمكن تصنيف هذه الأساليب بشكل عام إلى ثلاثة أصناف: أساليب صورية **formal**، وغير صورية، وشبه صورية.

تعتمد الأساليب الصورية مثل مخطط الحالات المنتهية **Finite State Diagram** ولغة **Z** على أساس رياضي متين ودقيق،

بينما لا يمكن توصيف الأساليب الغير صورية مثل مخططات التدفق بشكل رياضي متين ولا يمكن بالتالي دراستها تحليلياً.

إن كل ما يمكن عمله باستعمال الأساليب الغير صورية هو إيجاد مثال معاكس يبين أين فشل النظام في تحقيق احتياجاته أو أين حدث تعارض في النظام.

هذه الطريقة ليست مناسبة في نظم الزمن الحقيقي حيث نحتاج لتوصيف دقيق وواضح لاحتياجات أداء النظام.

تسمى الأساليب التي لا يمكن تصنيفها على أنها صورية تماماً أو غير صورية تماماً بالأساليب

شبه الصورية. ويمكن أن تعتمد بعض هذه الأساليب على توصيفٍ رياضي، رغم أنها تبدو كأنها ليست كذلك.

على سبيل المثال، يعتبر البعض أن لغة النمذجة الموحدة **Unified Modeling Language (UML)** هي شبه صورية لأن مخطط الحالة **Statechart** فيها هو صوري لكنه يستعمل تقانات نمذجة لها أساس شبه رياضي، بينما يعتبر البعض الآخر أن **UML** هي ليست حتى شبه صورية لأن توصيفها فيه ثغرات خطيرة وعدم تجانس،

لذلك جرى اقتراح تعديل على **UML** اسمه **UML2.0** يحوي مكونات لها توصيف صوري نظامي. وهناك محاولات أخرى لجعلها أكثر صورية.

4. الطرق الصورية لتوصيف الاحتياجات البرمجية

تحاول الطرق الصورية تحسين القدرة التعبيرية للاحتياجات بتمديد وتعديل طرقٍ رياضيةٍ معروفة، مثل منطق القضيات **propositional logic** وحساب الإسناديات **predicate calculus** ونظرية المجموعات **set theory**. وهي جذابة ومفيدة لأنها تقدم طريقةً علميةً ممنهجةً لتوصيف الاحتياجات، ويؤدي عادةً استعمالها لتوصيف الاحتياجات إلى اكتشاف الأخطاء التصميمية منذ المراحل الأولى لدورة تطوير البرمجيات، حيث يمكن تصحيحها بسرعة وبأقلّ كلفةٍ ممكنة.

لكن تتصف الطرق الصورية مع ذلك بأنها صعبة القراءة والاستعمال حتى من قبل الخبراء المدربين. لذلك يتفادى الكثيرون استعمال هذه الطرق لتوصيف احتياجات أنظمتهم ويستعملون عوضاً عنها طرقاً غير صورية أو شبه صورية.

هناك العديد من الطرق الصورية لتوصيف الاحتياجات منها:

- منطق القضيات **propositional logic**.
- حساب الإسناديات **predicate calculus**.
- لغة **Z**.
- آلة الحالات المنتهية **Finite State Machine**.
- مخططات الحالة **Statecharts**.
- شبكات بتري **Petri nets**.

وسنستعرض فيما يلي نموذجاً "آلة الحالات المنتهية" و "شبكات بتري" الأكثر استعمالاً في

تصميم نظم الزمن الحقيقي.

آلة الحالات المنتهية Finite State Machine (FSM)

وتسمى أيضاً "أوتومات الحالات المنتهية" **Finite State Automaton (FSA)** و "مخطط انتقـال الحالة" **State Transition Diagram (STD)**، وهو نموذج رياضي صوري يستعمل لتوصـيف وتصميم مجال واسع من النظم.

يمكن التعبير ببساطة عن هذه الآلة بأنها تعتمد على حقيقة أنه يمكن تمثيل معظم النظم باستعمال عدد محدد من الحالات المتمايزة. ويمكن للنظام أن ينتقل من حالة لأخرى بمـرور الـزمن أو بحدوث حدث معين.

يمكن التعبير عن FSM على شكل مخطط أو جدول.

لنفترض على سبيل المثال أننا نريد نمذجة نظام تحكم حاسوبي بطائرة مقاتلة. يحتوي النظام على وحدة قياس للعطالة، ويمكن لنظام التحكم أن يعمل في أحد الحالات الخمس التالية:

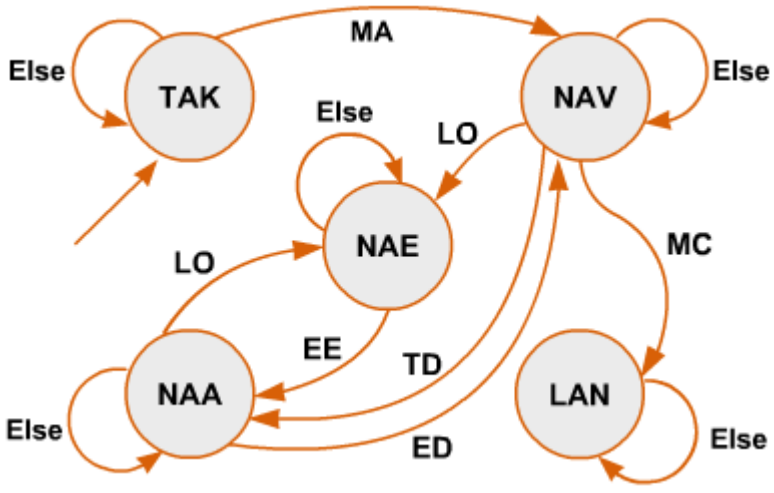
الإقلاع **TAK** والملاحة **NAV** والملاحة/هروب **NAE** والملاحة/هجوم **NAA** والهبوط **LAN**.

يستجيب النظام لعدد من الإشارات الواردة من أجزاء أخرى من الطائرة وهي:

إسناد المهمة **MA** وقفل العدو **LO** واكتشاف الهدف **TD** وإتمام المهمة **MC** وهروب العدو **EE** وتدمير العدو **ED**.

الحالة البدائية للنظام هي **TAK** والحالة النهائية الوحيدة هي **LAN**.

يبين الشكل التالي مخطط انتقال النظام من حالة لأخرى حسب الإشارات الواردة إليه:



يمكن التعبير عن نفس المخطط على شكل جدول انتقال حالة، أسطره هي الحالة الحالية، أعمدته هي إشارة الدخل، والقيم المخزنة فيه هي الحالة التالية. يبين الجدول التالي تمثل نفس **FSM** السابقة باستعمال جدول الحالة:

	MA	LO	TD	MC	EE	ED
TAK	NAV	TAK	TAK	TAK	TAK	TAK
NAV	NAV	NAE	NAA	LAN	NAV	NAV
NAE	NAE	NAE	NAE	NAE	NAA	NAE
NAA	NAA	NAE	NAA	NAA	NAA	NAV
LAN	LAN	LAN	LAN	LAN	LAN	LAN

كمثال آخر على آلة الحالات المنتهية، نفترض أن أي إجراء في نظام تشغيل الزمن الحقيقي يمكن أن يكون في أحد الحالات الثلاث التالية: مجمد **suspended** وجاهز **ready** وقيد التنفيذ **executing**.

يمكن بسهولة استنتاج الأحداث التي تجعل الإجراء ينتقل من حالة لأخرى من المخطط التالي:

تتصف **FSM** بأنها سهلة التطوير ويمكن تحقيقها بسهولة باستعمال جداول الانتقال. وهي كـذلك غير غامضة وقادرة على التعبير عن مفاهيم مهمة بأسلوب سهل. كـذلك تتـوفر الكثيـر مـن الدراسات والخوارزميات التي تمكن المصمم من اختزال **FSM** لأقصى حد.

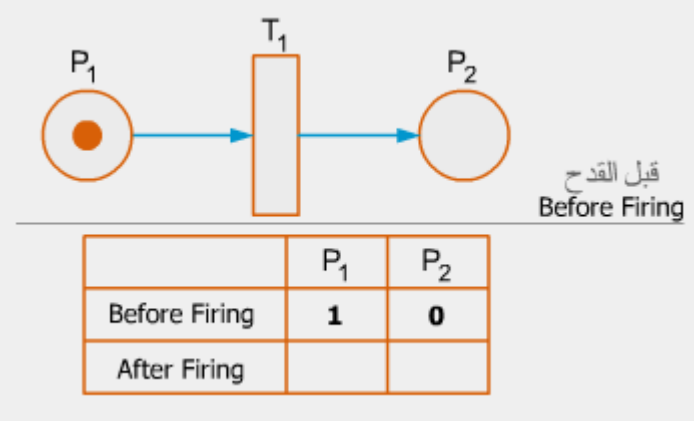
لكن يمكن أحياناً أن يكبر عدد الحالات لمقدار كبير يجعل من الصعب قراءتها والتعبيـر عنهـا بوضوح.

شبكات بتري

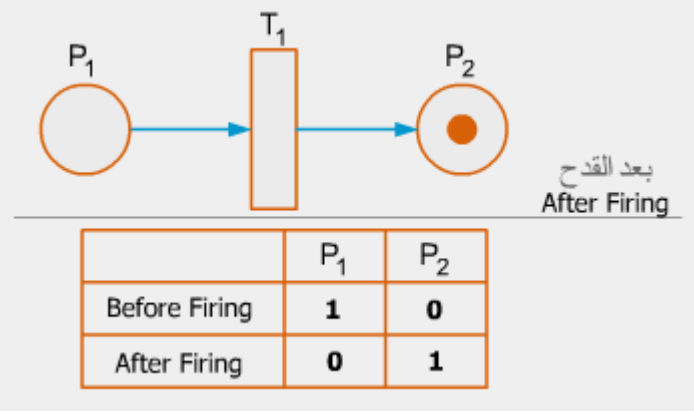
وهي طريقة صورية أخرى تستعمل لتوصيف العمليات التي يجب تأديتها في بيئة متعددة المعالجات أو متعددة الإجراءات.

يمكن التعبير عنها بأسلوب رياضي أو بأسلوب بياني.

تحتوي شبكة بتري على مجموعة من الدوائر تسمى "الأماكن" **places** وتمثّل الإجراءات أو مخازن المعطيات. تمثّل المستطيلات العمليات أو الانتقالات. يكتَب بجانب كل إجراء عدد المعطيات الموجودة فيه، وبجانب كل انتقال تابع الانتقال الموافق. ويصل بين الدوائر والمستطيلات أضلاع وحيدة الاتجاه.

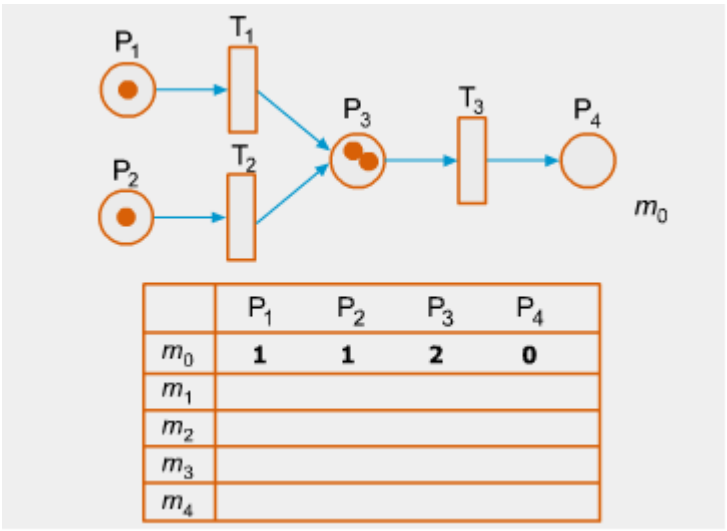


	P_1	P_2
Before Firing	**1**	**0**
After Firing		

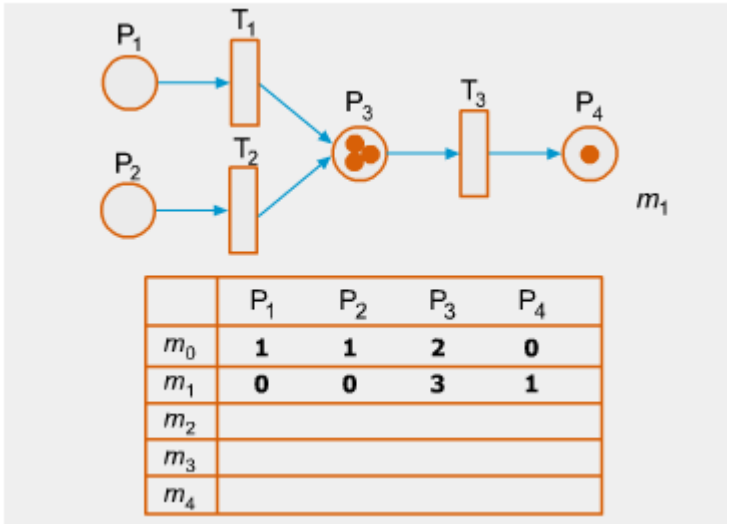


	P_1	P_2
Before Firing	**1**	**0**
After Firing	**0**	**1**

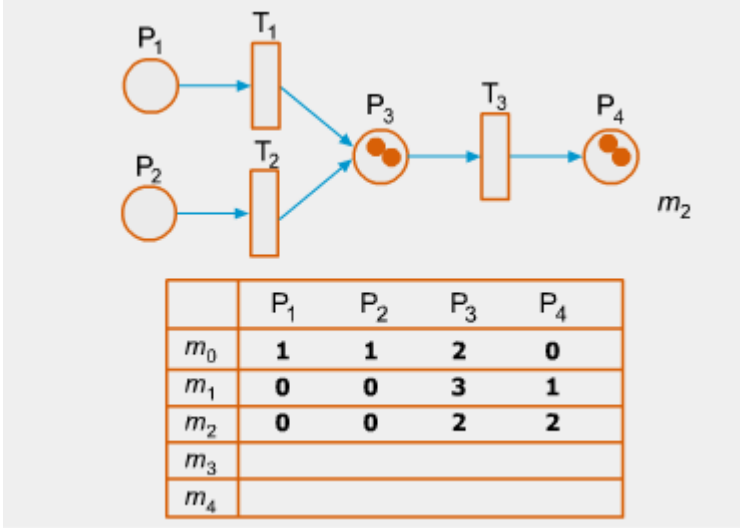
يبين الشكل التالي والجدول الذي يليه مثالاً آخر لشبكة بتري أعقد من السابقة:



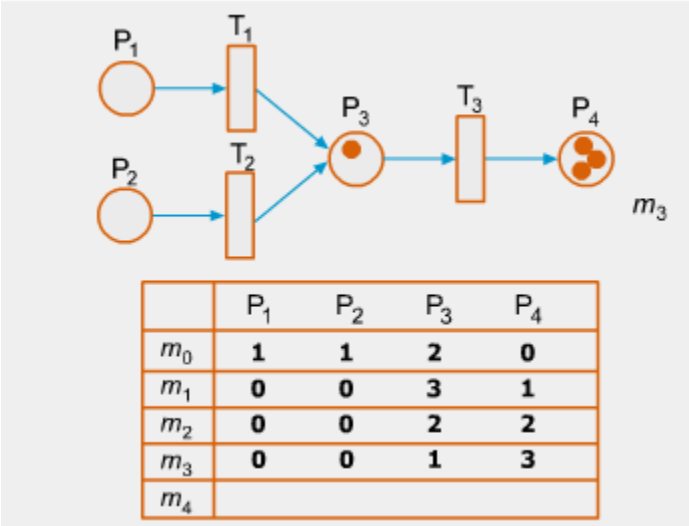
	P_1	P_2	P_3	P_4
m_0	**1**	**1**	**2**	**0**
m_1				
m_2				
m_3				
m_4				



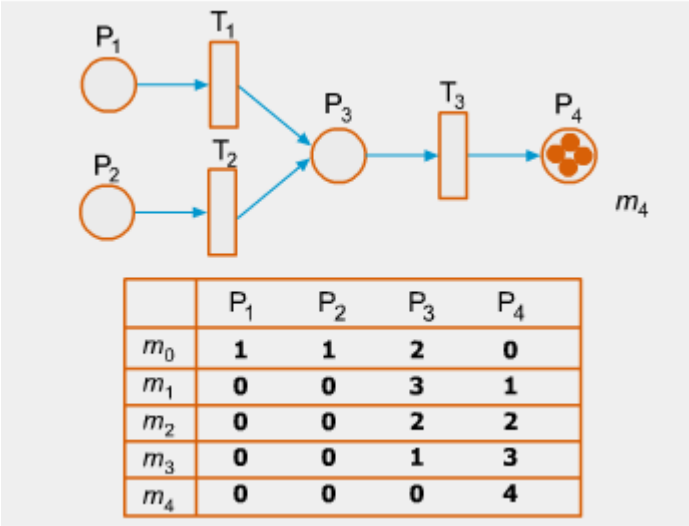
	P_1	P_2	P_3	P_4
m_0	**1**	**1**	**2**	**0**
m_1	**0**	**0**	**3**	**1**
m_2				
m_3				
m_4				



	P_1	P_2	P_3	P_4
m_0	**1**	**1**	**2**	**0**
m_1	**0**	**0**	**3**	**1**
m_2	**0**	**0**	**2**	**2**
m_3				
m_4				



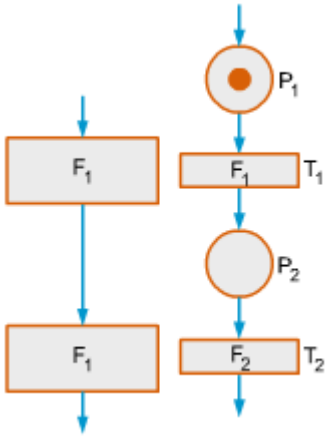
	P_1	P_2	P_3	P_4
m_0	1	1	2	0
m_1	0	0	3	1
m_2	0	0	2	2
m_3	0	0	1	3
m_4				



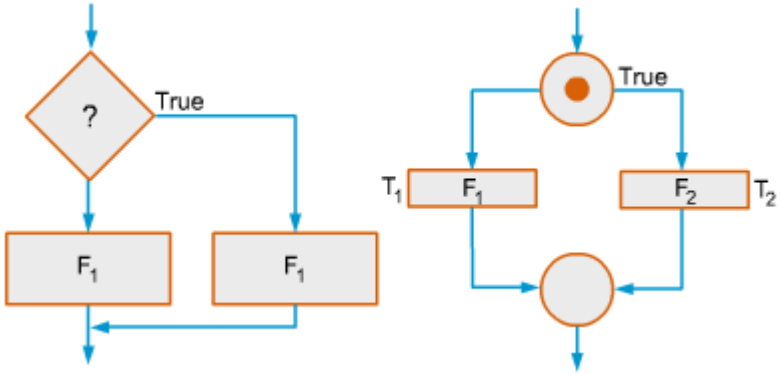
	P_1	P_2	P_3	P_4
m_0	1	1	2	0
m_1	0	0	3	1
m_2	0	0	2	2
m_3	0	0	1	3
m_4	0	0	0	4

يمكن لشبكات بتري أن تستعمل لنمذجة النظم وتحليل القيود الزمنية وحالات التسابق **race conditions**، كما يمكنها نمذجة البنى المعروفة لمخططات التدفق **flowcharts** كما هو مبين في الأشكال التالية:

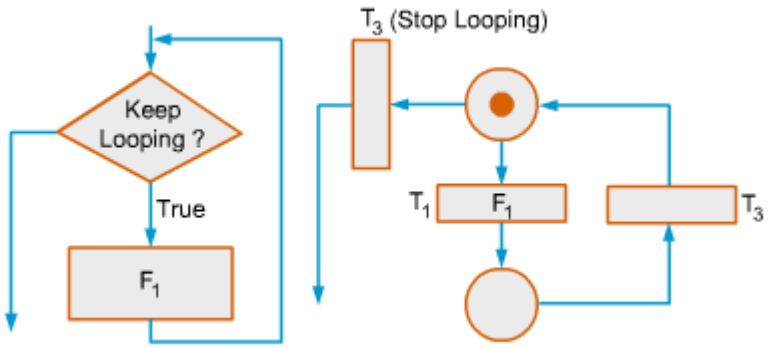
يبين الشكل التالي كيفية الحصول على التسلسل **sequence** باستعمال شبكات بتري:



أما الشكل التالي فيبين كيفية الحصول على التفريع الشرطي:



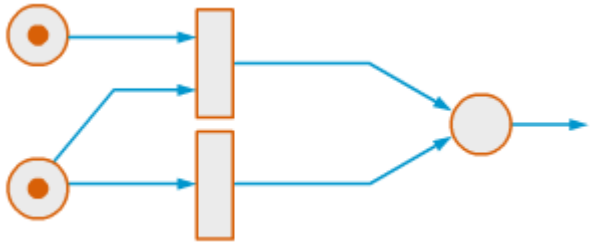
ويعبر الشكل التالي عن حلقة **while**:



تفيد شبكات بتري كثيراً عند الحاجة لتمثيل النظم المتعددة المعالجات أو المتعددة الإجراءات. هناك أيضاً عدد من التمديدات التي يمكن تطبيقها على هذا النموذج للحصول على نماذج أكثر قوة مثل شبكات بتري الموقوتة وشبكات بتري الملونة وشبكات بتري الملونة والموقوتة.

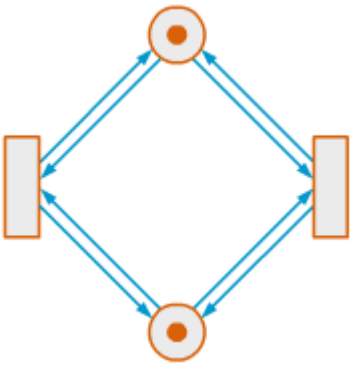
إن شبكة بتري هي أداة قوية يمكن استعمالها أثناء تحليل أو تصميم النظام لكشف حالات القفل المتبادل **dead-lock** وحالات التسابق.

لنفرض على سبيل المثال أن جزءاً من شبكة بتري لنظام ما هو المبين في الشكل التالي:



من الواضح أنه في هذه الحالة لا يمكن معرفة أي الانتقالين سيقدح.

يمكن كذلك استعمال شبكات بتري لكشف الدورات في النظام التي يمكن أن تؤدي لقفل متبادل كما هو مبين في الشكل التالي:



من الواضح أيضاً أن هذه الحالة هي حالة قفل متبادل.

ويمكن أخيراً استعمال شبكات بتري لكشف وجود حلقات أكبر وأقل وضوحاً من هذا المثال وتضم عدداً كبيراً من الإجراءات.

مقرر: نظم الزمن الحقيقي

الفصل الرابع: لغات برمجة نظم الزمن الحقيقي

فهرس الفصل:

1. مقدمة

يسبب سوء استعمال لغة البرمجة المستعملة لبرمجة نظام الزمن الحقيقي أثراً كبيراً على أداء النظام ككل، ويسبب بالتالي تعدي العديد من الإجراءات لحدودها الزمنية.

كذلك فإن الاستعمال المتزايد للغات الغرضية التوجه **Object-Oriented Programming Languages** مثل **Java** و **C++** و **ADA** تجعل المشكلة أكثر حرجاً وصعوبة. لكن لا يمكن إنكار أن لهذه اللغات العالية المستوى مزايا عديدة جعلت مبرمجي نظم الزمن الحقيقي يستعملونها بكثرة عوضاً عن اللغات المنخفضة المستوى مثل **C** ولغة التجميع **Assembly Language**.

إن البرمجة هي أقرب إلى "الحرفة" و "الفن" منها إلى الإنتاج الكمي. ومثل أية حرفة، تحتاج البرمجة إلى الأدوات الضرورية والملائمة ذات الجودة العالية، وتحتاج لأن يتقن المبرمج العمل عليها. الأداة الأساسية التي يستعملها المبرمج لتطوير البرمجيات هي المترجم **Compiler**.

استعملت العديد من لغات البرمجة لبناء نظم الزمن الحقيقي مثل **C** و **C++** و **C#** و **Java** و فورتران وباسكال و **ADA95** ولغة التجميع، وحتى **Visual Basic** و **Lisp** و **Prolog**. بعض هذه اللغات غرضية التوجه مثل **C++** و **Java** و **C#**، وبعضها إجرائية **procedural** مثل باسكال وفورتران. وتمتلك لغة **ADA95** عناصر من الطرفين ويمكن استعمالها كلغة إجرائية أو غرضية التوجه حسب رغبة المبرمج. إن استعمال بعض اللغات الخاصة مثل **Lisp** و **Prolog** في برمجة نظم الزمن الحقيقي غير شائع، وإن كان موجوداً.

السؤال المهم هنا هو: "كيف يمكن معرفة مدى ملائمة لغة ما لبرمجة نظم الزمن الحقيقي، وما هو المعيار الذي يمكن استعماله لقياس ذلك؟" يمكن الإجابة على هذا السؤال بشكل تقريبي عبر المعايير التجريبية التالية المسماة "**معايير كارديللي**" **Cardelli's Criteria**:

- **الاقتصاد في زمن التنفيذ:** ما هي السرعة التي يعمل بها البرنامج الناتج؟
- **الاقتصاد في زمن الترجمة:** ما هو الزمن الناتج لتوليد البرنامج التنفيذي انطلاقاً من البرنامج المصدر؟
- **الاقتصاد في التطوير على نطاق صغير:** مقدار الجهد الذي يجب على مبرمج وحيد أن يبذله لتطوير النظام.
- **الاقتصاد في التطوير على نطاق واسع:** مقدار الجهد الذي يجب على مجموعة من المبرمجين أن يبذلوه لتطوير النظام.

- **الاقتصاد من ناحية ميزات اللغة:** ما هو مقدار صعوبة تعلم ميزات **features** لغة البرمجة؟

لكل لغة برمجة نقاط قوة وضعف عندما يُراد استعمالها لبرمجة نظم الزمن الحقيقي، ويمكن استعمال المعايير السابقة لمقارنتها مع بعضها البعض واختيار الأنسب منها حسب التطبيق المطلوب والمهارات البرمجية المتاحة لدى فريق التطوير.

سنقدم في هذا الفصل عرضاً للغات البرمجة التي يمكن استعمالها لبرمجة نظم الزمن الحقيقي، بالإضافة إلى الأساليب الواجب اتباعها لتحسين البرنامج الناتج قدر الإمكان بحيث يصبح أمثلياً وقابلاً للتنبؤ من ناحية الأداء والتصرف.

2. لغة التجميع Assembly

على الرغم من انخفاض مستوى هذه اللغة وعدم احتوائها على العديد من المزايا التي تقدمها اللغات عالية المستوى، تمتلك لغة التجميع بعض المزايا التي تجعلها ملائمة في بعض الأحيان لبرمجة نظم الزمن الحقيقي. فهي تسمح للمبرمج بالتحكم المباشر والمطلق بالبنيان الصلب للنظام، ويمكن أن يكون البرنامج المكتوب مباشرةً بلغة التجميع أكثر أمثليةً من البرنامج الناتج عن مترجمات اللغات عالية المستوى. لكن التطور الكبير الذي حصل في مجال تصميم وكتابة المترجمات جعل برنامج التجميع الناتج عنها أمثلياً إلى حد كبير بحيث يصعب على معظم المبرمجين كتابة برامج أفضل منه بلغة التجميع مباشرة.

تعاني أيضاً لغة التجميع من أنها غير مهيكلة وذات تجريد ضعيف جداً. وهي تختلف كثيراً من معالج لآخر والبرنامج الناتج عنها ذو ناقلية **portability** ضعيفة. كذلك فإن البرمجة بلغة التجميع صعبة ومجهدة ومليئة بالأخطاء البرمجية. لذلك لا يُنصح باستعمالها على الإطلاق. يقتصر استعمالها عادةً في الحالات التي لا يقدم فيها مترجم اللغة عالية المستوى بعض التعليمات أو البنى الضرورية لعمل النظام، أو عندما تكون القيود الزمنية المفروضة على النظام حرجة جداً بحيث يجب أن يُكتَب بلغة الآلة ليكون أمثلياً ومُتنبأ به تماماً.

كحل وسط، يمكن كتابة[1] الأجزاء الحساسة فقط من النظام بلغة التجميع وكتابة الباقي بلغة عالية

[1] تسمح بعض اللغات عالية المستوى مثل ADA95 وباسكال بحشر أجزاءٍ مكتوبة بلغة التجميع ضمن البرنامج.

المستوى.

إذا أردنا تقييم لغة التجميع حسب معايير كارديللي، يمكن القول أنها اقتصادية جـداً فـي زمـن التنفيذ والترجمة (لأنها غير مترجمة أصلاً!)، لكنها غير اقتصادية في التطوير على نطاق ضيق أو واسع، وفقيرة من ناحية ميزات اللغة.

3. اللغات الإجرائية Procedural Languages

وهي اللغات التي تُعرف فيها عمليات البرنامج على شكل لائحة من التعليمات التسلـسلية تُنفـذ على التتابع. من الأمثلة عليها نذكر: **C**، باسـكال، فـورتران، **BASIC**، **ADA95**، **Modula-2**. يمكن في جميع هذه اللغات تجميع التعليمات في وحدات منطقية لها مهام محددة تسمى التوابـع **functions** أو البرامج الجزئية **sub-programs**. هناك العديد من الميزات التـي مـن المهـم توفرها في هذا النوع من اللغات لجعلها مناسبة أكثر لبرمجة نظم الزمن الحقيقي نذكر منها:

- تنوع أساليب تمرير المعاملات **parameters passing**
- تسهيلات الحجز الديناميكي للذاكرة **dynamic memory allocation**
- التنميط القوي للمتحولات **strong variae typing**
- أنماط المعطيات المجردة **abstract data typing**
- معالجة الاستثناءات (أخطاء زمن التنفيذ) **exceptions handling**
- توفر الوحدات البرمجية **modularity**

سنناقش الآن معايير كارديللي للغات الإجرائية.

يؤدي التنميط القوي للمتحولات إلى تحسين عملية توليد برنامج التجميع الناتج، مما يجعل اللغات الإجرائية اقتصادية جداً من ناحية زمن التنفيذ (بفرض أن المترجم جيد). كـذلك يـؤدي تـوفر الوحدات البرمجية في اللغة إلى إمكانية ترجمة كل وحدة على حدة، مما يجعل اللغات الإجرائية اقتصادية أيضاً في زمن الترجمة ويسهل عملية ترجمة الـنظم الكبيـرة بفـرض أن واجهـات **interfaces** الوحدات ثابتة إلى حد ما.

تتصف هذه اللغات أيضاً بأنها اقتصادية في التطوير على نطاقٍ صغير، لأن عملية التحقق مـن الأنماط الموجودة في مترجماتها تكشف معظم أخطاء البرمجة وتخفف مـن جهـود الاختبـار والتنقيح. وكذلك فإن تجريد المعطيات وتوفر الوحدات البرمجية لهما تأثير كبيـر فـي تـسهيل التطوير على نطاقٍ واسع. يمكن لفريق كبير من المبرمجين الاتفاق على الواجهات بين الوحدات

البرمجية، ثم تطوير كل منها واختبارها بشكلٍ مستقل، شرط تقليل البنى والعمليات العامة **global** واعتمادية الوحدات على بعضها البعض ما أمكن.

أخيراً، تتصف اللغات الإجرائية بالاقتصاد من ناحية ميزات اللغة، حيث لا تتصف بالتعقيد ويسهل عادةً تعلمها والإلمام بميزاتها المختلفة.

4. اللغات غرضية التوجه Object-Oriented Languages

لهذه اللغات مزايا عديدة معروفة ومفيدة جداً مثل الوثوقية **reliability** وزيادة فعالية المبرمج وإمكانية إعادة الاستعمال **reusability**.

يمكن تعريف هذه اللغات بأنها لغات البرمجة عالية المستوى التي تدعم تجريد المعطيات **data abstraction** والوراثة **inheritance** وتعددية الأشكال **polymorphism** وإرسال الرسائل **messaging**. ومن الأمثلة على هذه اللغات نذكر **C++** و **Java** و **C#** و **ADA95** و **Eiffel**.

إن الأغراض **objects** هي وسيلة فعالة لتخفيف تعقيد النظام لأنها تؤمن بيئة طبيعية لإخفاء المعطيات وكْبسلتِها **encapsulation**، حيث تُجمع معطيات الغرض والعمليات[2] التي تتم عليه في كتلةٍ واحدة سهلة الاستعمال. عادةً ما يكون استعمال الأسلوب الغرضي التوجه لتوصيف نظامٍ ما سهلاً وطبيعياً أكثر من الأساليب الأخرى.

تدعم لغات البرمجة غرضية التوجه الحديثة العديد من التقنيات المهمة في نظم الزمن الحقيقي وذلك من أساس اللغة، مثل النياسب **threads** والمزامنة **synchronization** والسلسلة **serialization**.

لنناقش الآن معايير كارديللي للغات الغرضية التوجه.

فيما يتعلق بالاقتصاد في زمن التنفيذ، تتصف هذه اللغات بأنها أسوأ من اللغات الإجرائية بطبيعتها وذلك لعدة أسباب أهمها بطء استدعاء الطرائق بسبب مستوى الوصول الغير مباشر الإضافي **indirection** (كلّ غرضٍ يحوي جدولاً بعناوين طرائقه، يجب الرجوع إليه عند كل استدعاءٍ لأحدها).

فيما يتعلق بالاقتصاد في زمن الترجمة، فإن اللغات غرضية التوجه أسوأ من اللغات الإجرائية

---

[2] إن هذه العمليات تسمى الطرائق methods، وهي تقابل التوابع في اللغات الإجرائية.

من هذه الناحية أيضاً. إذ أن مترجماتها معقدة وعادةً ما يكون من الصعب تقسيم النظم الكبيرة إلى وحداتٍ مستقلةٍ عن بعضها البعض. على سبيل المثال، إذا أدخل المبرمج تعديلاً على صف **class** ما، يجب عندها إعادة ترجمة كل الصفوف الموروثة منه. لذلك فإنه من الممكن أن يزيد الزمن اللازم لترجمة النظام بمعدلٍ كبيرٍ عندما يصبح حجمه كبيراً.

بالمقابل فإن اللغات غرضية التوجه أفضل من اللغات الإجرائية من ناحية الاقتصاد في التطوير على نطاق صغير لأنه يمكن للمبرمجين المنفردين الاستفادة من مكتبات الصفوف وبيئات العمل المتوفرة بكثرة لتخفيض جهد وزمن التطوير إلى حد كبير. ومع ذلك، يحتاج المبرمجون في كثير من الأحيان إلى الدخول في تفاصيل مكتبات الصفوف لفهم عملها واستعمالها بفعالية، وهو أمر أصعب عادة من الدخول في تفاصيل الوحدات البرمجية الإجرائية.

أما من ناحية الاقتصاد في التطوير على نطاق واسع، فتتفوق فيها اللغات الإجرائية على اللغات غرضية التوجه على الرغم تميز الأخيرة بصفة "إعادة الاستعمال" **reusability**. يعود السبب في ذلك إلى افتقارها للإجتزائية **modularity** بسبب صعوبة تعديل الصفوف باستعمال الوراثة **inheritance**. على سبيل المثال، يمكن أن يقوم المبرمج عن طريق الخطأ بإعادة تعريف **override** بعض التوابع الموروثة التي يجب عدم إعادة تعريفها لكي يعمل الصف جيداً. ولذلك يضطر المبرمج في كثير من الأحيان إلى الدخول في تفاصيل الصف الأصلي لكي يتمكن من تعديله باستعمال الوراثة.

وأخيراً، تتصف اللغات غرضية التوجه بضعف الاقتصاد في ميزات اللغة. على سبيل المثال، اعتمد تصميم لغة **C++** على نموذج بسيط نسبياً مبني على لغة **Simula**، لكنها مليئة بالميزات التي يصعب تعلمها واستعمالها بالشكل الصحيح. حتى اللغات الحديثة مثل جافا هي أعقد بكثير مما يظنه الكثير من الناس. لسوء الحظ، تبين أن مبدأ "كل شيء هو غرض **object**" الذي بدا اقتصادياً ومنتظماً في البداية، انتهى به الأمر ليصبح مجموعة هائلة من الصفوف المتعلقة ببعضها البعض والتي يصعب استعمالها ما لم تُفهم بمجملها.

5. لمحة عامة عن لغات برمجة الزمن الحقيقي

سنستعرض فيما يلي عدداً من اللغات المستعملة عادةً لبرمجة نظم الـزمن الحقيقـي بالترتيـب الأبجدي لأسمائها، وهي:

1. لغة **ADA95**.
2. لغة **C**.
3. لغة **C++**.
4. لغة **C#**.
5. فورتران **Fortran**.
6. لغة جافا **Java**.
7. لغة **Occam 2**.
8. لغات الزمن الحقيقي المخصصة.

حذفنا من هذا العرض اللغات التابعية **functional** مثل **Lisp** و **ML** وذلك لندرة استعمالها فـي نظم الزمن الحقيقي، وليس لقلة أهميتها. وحـذفنا كـذلك بعـض اللغـات الخطاطيـة **script languages** مثل **Python** و **Ruby**، ذلك لأنها غير ملائمةٍ عادةً للنظم المضمنة.

لغة ADA95

كان الهدف الأساسي لتطوير لغة **ADA** هو أن تصبح اللغة الإجبارية لتطوير كافة مشاريع إدارة الدفاع الأمريكية والتي تتضمين الكثير من النظم المضمنة. لكن النسخة القياسية الأولـى التـي ظهرت عام **1983** احتوت على الكثير من المشاكل التي أصبحت معروفةً للمبرمجين منذ ظهور أول مترجمٍ لها. لذلك قام مطوروها بحل هذه المشاكل لتظهر النسخة الثانية المـسماة **ADA 95** والتي تُعتبر أول لغةٍ غرضية التوجه قياسيةٍ عالمية. لكن إدارة الدفاع وافقت على استعمال لغاتٍ أخرى في العديد من المشاريع، مما جعل المطورين يدركون أن لغة **ADA** غير مناسـبةٍ لتلبيـة الهدف الأساسي منها.

صُممت لغة **ADA** خصيصاً لتطبيقات نظم الزمن الحقيقي المضمنة. لكن المطورين وجدوا أنهـا لغة ثقيلة وغير فعالةٍ لمثل هذه التطبيقات، بالإضافة إلى احتوائها على العديد من المشاكل عنـد تحقيق النظم المتعددة المهام باستعمال الأدوات القليلة التي تتيحها اللغة.

أخيراً، على الرغم من أن عدد المطورين الذين يستعملون لغة **ADA** يتناقص باسـتمرار، إلا أن

توفر نسخةٍ مفتوحة المصدر من لغة **ADA** على نظام لينوكس أعطاها دفعاً صغيراً مؤخراً.

لغة C

أختُرعت لغة **C** عام **1971** وهي لغة مناسبة للتطبيقات "المنخفضة المستوى". ويعود السبب في ذلك إلى أنها مشتقة من لغة **BCPL** التي كان النمط الوحيد الذي تدعمه هو "كلمة الآلة" **machine word**.

تدعم لغة **C** الأنماط الأساسية منخفضة المستوى مثل البايتات والبتات والمحارف والعناوين التي يمكن استعمالها للتعامل مع متحكمات المقاطعات **interrupt controllers** وسجلات المعالج والتجهيزات الأخرى الضرورية في نظم الزمن الحقيقي. لذلك، غالباً ما تُستعمل لغة **C** كلغة تجميع عالية المستوى مستقلةٍ عن البنيان المادي.

تحتوي لغة **C** على العديد من أنماط المتحولات التي تسمح بالتحكم بتوليد الترميز النهائي من مستوى عالٍ مثل **register** و **volatile** و **static** و **constant**.

على سبيل المثال، إذا عُرف متحول بالنمط **register**، فهذا يدل على أنه سيُستعمل كثيراً، مما يدعو المترجم **compiler** لتخصيص سجل من سجلات المعالج لهذا المتحول مما يجعل ناتج الترجمة أسرع وأصغر حجماً. كذلك، تخبر الصفة **volatile** المترجم بألا يحاول أمثلة **optimize** المتحولات المعرفة بهذه الصفة. هذا الأمر مفيد على سبيل المثال للتعامل مع الدخل/خرج المعنون بالذاكرة.

بشكلٍ عام، تُعتبر لغة **C** مناسبةً للبرمجة المضمنة لأنها لغة مهيكة ومرنة وخالية من القيود.

لغة C++

لغة **C++** هي لغة غرضية التوجهة هجينة (أي أنها لغة إجرائية أيضاً) بدأت كطبقة ماكروية مبنية على لغة **C**، لكنها أصبحت اليوم لغةً مستقلةً لها مترجماتها الخاصة المتوافقة مع **C**. تمتلك **C++** جميع مزايا اللغات غرضية التوجهة وتؤمن كبسلةً وتجريداً أفضل من **C**. يتفق معظم المبرمجون على أن إساءة استعمال المؤشرات هو المصدر الأساسي للأخطاء البرمجية في **C/C++**. فقد استعمل مبرمجو **C/C++** الأوائل عمليات حسابية معقدة على المؤشرات لإنشاء ومعالجة بنى المعطيات المعقدة، ولاسيما سلاسل المحارف **strings**. لذلك، كانوا يقضون وقتاً طويلاً لاصطياد أخطاء برمجية معقدة ناتجة عن معالجة بسيطة لسلاسل المحارف. حالياً، تحتوي **C++** على العديد من المكتبات النظامية لبنى المعطيات الديناميكية مثل **Standard**

Template Library (STL) والتي تحتوي على الأنماط **string** و **wstring**، مما يريح المبـرمج من عناء التعامل مع سلاسل المحارف.

يتزايد كل يوم عدد النظم المضمنة المبرمجة باستعمال **C++** ويسأل المبرمجون أنفسهم كل يوم: هل أستعمل **C** أم **C++** لتحقيق نظامي المضمن؟ الجواب هو: الأمر يعتمـد علـى مـا يريـده المبرمج.

إن اختيار **C** بدلاً من **C++** يعني الحصول على برنامج سريع ويسهل التنبؤ بتـصرفاته، لكنـه صعب الصيانة. وبالعكس، فإن البرامج المكتوبة بلغة **C++** أبطأ ويصعب التنبؤ بتصرفاتها، لكن صيانتها أسهل بكثير من البرامج المكتوبة بلغة **C**.

لغة C#

وهي لغة شبيهة بلغة **Java** ولها بيئة تشغيل تشبه كثيراً آلة جافـا الافتراضـية **Java Virtual Machine (JVM)** وتسمى "منصة عمل **.NET**".

لكنها على عكس جافا، تسمح بتعليمات "غير آمنة" **unsafe code** يمكـن فيهـا التعامـل مـع المؤشرات مباشرةً وإجراء العمليات عليها كما في لغة **C/C++**.

على الأغلب، لا تناسب لغة **C#** وبيئة التشغيل **.NET** تطبيقات نظم الزمن الحقيقي الصعبة لعـدة أسباب، منها زمن التنفيذ الغير محدود لجامع النفايات **Garbage Collector** الذي تعتمـد عليـه بيئة التشغيل، وعدم توفر البنى اللازمة لجعل جدولة النياسب **threads** حتمية.

لكن من الممكن أن تناسب **C#** تطبيقات نظم الزمن الحقيقي الرخوة والقاسـية وذلـك لقابليتهـا لاستدعاء خدمات نظام التشغيل **operating system API** بشكل يحجب تعقيد إدارة الذاكرة عن المبرمج، بالإضافة إلى أن أداء عمليات الفاصلة العائمة فيها يقارب أداء لغة **C**.

فورتران Fortran

تُعتبر لغة فورتران من أقدم اللغات عالية المستوى المستعملة لبرمجة نظـم الـزمن الحقيقـي. وبسبب افتقاد النسخ الأولية من فورتران للعودية **recursion** والحجز الديناميكي للـذاكرة، فقـد احتوت النظم المكتوبة بها على أجزاء كبيرة مكتوبة بلغة التجميع لمعالجة المقاطعـات وجدولـة الإجراءات. أما الاتصال مع الأجهزة الخارجية، فكان يجري باستعمال **DMA** والـدخل/خـرج المعنون بالذاكرة **Memory-mapped I/O** وتعليمات الدخل/خرج.

تحتوي النسخ الحديثة من فورتران على دعم للعودية. ورغم ذلك، مازالت نظم الزمن الحقيقـي

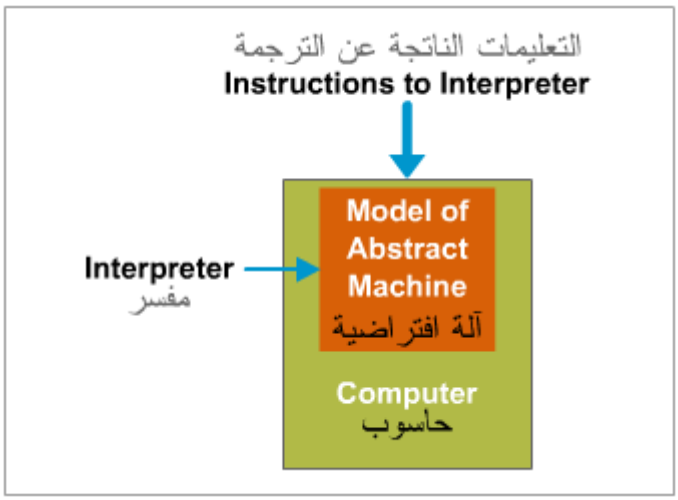
الحالية المكتوبة بها تحتوي على أجزاء مكتوبة بلغة التجميع.

طورت فورتران في وقت كان من الضروري فيه الحصول على أداء أمثلي على أجهزة حاسوب صغيرة الحجم وبطيئة. ونتيجة لذلك، كانت البنى الأساسية للغة منتقاةً لهدف واحـد هـو الأداء، وهو ما كان محققاً بالفعل في أغلب الأحيان.

تتلخص مساوئها بأنها ضعيفة التنميط **weakly-typed** وتفتقر لمعالجة أخطاء زمن التنفيذ وأنماط المعطيات المجردة. ولكنها ما زالت تُستَعمل حتى الآن في العديد مـن نظـم الـزمن الحقيقـي الحديثة.

لغة جافا Java

جافا هي لغة مفسرة. يحول مترجم جافا البرنامج المصدر إلى لغة آلة وسيطة مستقلة عـن أي معالج اسمها **byte code**، ويقوم مفسر خاص هو آلة جافا الافتراضية **Java Virtual Machine** **JVM** بتنفيذ هذه التعليمات في بيئة مُدارة **managed** كما هو مبين في الشكل التالي:



ميزة هذا الأسلوب هو الناقلية **portability**، حيث يمكن لبرنامج جافا المترجم أن يعمل على أية آلة يتوفر عليها آلة جافا الافتراضية. وهناك العديد من التطبيقات الهامة لهذا الأسلوب في النظم المضمنة والمحمولة، مثل الهواتف الخلوية والبطاقات الذكية والويب.

ويتوفر أيضاً مترجمات حقيقية للغة جافا ينتج عنها ترميز ثنائي يعمل على نـوع خـاص مـن المعالجات كغيرها من اللغات المترجمة الأخرى مثل **C++**. يتوفر كذلك معالجات قـادرة علـى تنفيذ تعليمات **byte code** مباشرة.

جافا هي لغة غرضية التوجه تشبه كثيراً لغة **C++**، لكنها لا تعتمد مفهـوم المعالجـة الأوليـة

للبرنامج **preprocessor**. وهي كذلك لا تدعم التمرير بالمرجع **call by reference** وليس فيها مؤشرات صريحة. جميع الأغراض والمصفوفات فيها هي مؤشرات ضمنية ويجـب حجزهـا ديناميكياً، وتقع مسؤولية تحريرها على كاهل جزء خاص من المفـسر هـو جـامع النفايـات **Garbage Collector**. كذلك فإن المفسر يختبر معظم الحالات التي يمكن أن تولّد أخطاءاً فـي زمن التنفيذ مثل خروج دليل مصفوفة خارج المجال الصحيح أو التقسيم على صفر.

كل هذا يجعل أداء لغة جافا أسوأ من أداء اللغات المترجمة الأخرى مثل **C++**.

لغة real-time java

مشكلة لغة جافا الأصلية هو الأداء الذي لا يمكن التنبؤ به للبرنامج بسبب جامع النفايـات ولأن المواصفات الأصلية للغة تحدد خطوطاً عريضة فقط للجدولة.

على سبيل المثال، عند تنافس عدة نياسب للوصول إلى مصادر مشتركة، تُنفذ عادةً النياسب ذات الأولويات العالية أولاً على حساب النياسب ذات الأولويات المنخفضة. لكن هـذا لا يـضمن أن تبقى النياسب ذات الأولويات العالية في حالة تنفيذ، كما لا يمكن الاعتماد على أولويات النياسب لتحقيق المنع المتبادل بشكل موثوق. كل هذا جعل المطورين يعلمون أن لغة جافا القياسية غيـر مناسبة لتطبيقات الزمن الحقيقي.

وللتغلب على هذه المشاكل، كُلف المعهد الوطني للمقاييس والتكنولوجيـا **National Institut of Standards and Technology (NIST)** بتطوير نسخة من جافا مناسبة لتطبيقات الزمن الحقيقي والنظم المضمنة. نُشِر تقرير مجموعة العمل المكلفة بهذا الأمر في شـهر أيلـول عـام **1999**، وعرف تسعة احتياجات أساسية لمواصفات نسخة الزمن الحقيقي من لغـة جافـا **Real-Time Specification for Java (RTSJ)** وهي:

1. يجب أن تتضمن المواصفات إطار عمل لاكتشاف والبحث عن أية محللات **profilers** متاحة.
2. يجب أن يكون جامع النفايات ذو تأخير شُفعي محدود.
3. يجب أن تعرف المواصفات العلاقات بين نياسب الزمن الحقيقي بنفس مستوى التفـصيل المتاح حالياً في وثائق المعايير الأخرى.
4. يجب أن تتضمن المواصفات واجهات **API** لاتصال ومزامنة الإجراءات المكتوبة بلغـة جافا مع الإجراءات المكتوبة باللغات الأخرى.
5. يجب أن تتضمن المواصفات أسلوب معالجة الأحداث الداخلية والخارجية الغير متزامنة.

6. يجب أن تتضمن المواصفات إمكانية الإنهاء الغير متزامن للنياسب.
7. يجب أن تتضمن المواصفات أساليب لتحقيق المنع المتبادل دون تجميد.
8. يجب أن تتضمن المواصفات أساليب تمكّن تعليمات البرنامج من معرفة هل يجري تنفيذها في نيسب زمن حقيقي أم في نيسب عادي.
9. يجب أن تعرف المواصفات العلاقات بين نياسب الزمن الحقيقي والنياسب العادية.

تحقيق لغة Real-time Java

تعرف المواصفات **RTSJ** صفاً **RT** يعبر عن نيسب زمن حقيقي، ويقوم مجدول مقيم خاص بجدولتها وفقاً لمفاهيم الزمن الحقيقي. ويمكن لنياسب **RT** الوصول إلى الأغراض في الكومة **heap**، لذلك فهي عرضة للتأخيرات الزمنية التي يسببها جامع النفايات. فيما يتعلق بجمع النفايات، يعرف **RTSJ** نموذج ذاكرة موسع يدير الذاكرة بطريقة لا تؤثر على إمكانية نياسب الزمن الحقيقي بتحقيق التنفيذ الحتمي **deterministic**. يسمح هذا النموذج لكل من الأغراض ذات الأعمار القصيرة والطويلة بالتواجد خارج منطقة الكومة التي يتحكم بها جامع النفايات. وهو مرن كفاية بحيث يسمح بحلول معروفة لإدارة الذاكرة بفعالية مثل حوض الأغراض المحجوزة مسبقاً **pre-allocated object pool**.

تستعمل **RTSJ** مفهوم أولويات أكثر "رخاوة" مما هو مقبول عادةً في نظم الزمن الحقيقي. تدل الأولوية العالية لإجراءٍ ما على أنه المؤهل الأكبر لينتقيه المجدول من بين الإجراءات الجاهزة للتنفيذ، ولا تدل أبداً على أن عملية التوزيع **dispatching** تجري بأسلوب يعتمد على الأولويات.

يقوم النظام بتخزين جميع النياسب التي تنتظر امتلاك مصدر ما في رتل أولوية (أي مرتبة في الرتل حسب أولوياتها). تشمل هذه المصادر المعالج ومناطق الذاكرة المشتركة. يقدم **RTSJ** للمبرمج صفان يمكنانه من الوصول المباشر للذاكرة الفيزيائية:

1. **RawMemoryAccess**: الذي يعرف توابع تسمح ببناء أغراض تمثّل مجالات في الذاكرة الفيزيائية، والوصول إليها على مستوى **byte** و **word** و **long**.
2. **PhysicalMemory**: الذي يسمح بالحصول على أغراض من **PhysicalMemoryArea** الذي يعبر عن مجال من عناوين الذاكرة الفيزيائية التي يمكن للنظام أن يحجز الأغراض فيها.

لغة Occam 2

تعتمد لغة **Occam 2** على نموذج "الإجراءات التسلسلية المتخاطبة" **Communicating Sequential Processes (CSP).**

الوحدة الأساسية فيها هي "الإجراء" الذي يوجد منه أربعة أنواع أساسية: الإسناد، الدخل، الخرج والانتظار. وتُبنَى الإجراءات الأعقد بتركيب تسلسلي أو تفرعي من هذه الإجراءات.

البنية الشرطية الأساسية هي **IF** التي تحوي قائمةً من شروط وإجراءات موافقة لها.

صُممت لغة **Occam 2** أصلاً لدعم المعالجة التفرعية على الأجهزة التفرعية من نوع **transputers**، لكن يوجد لها أيضاً مترجمات للبنى الأخرى.

أخيراً، استُعمِلت هذه اللغة في العديد من التطبيقات في بريطانيا خصوصاً.

لغات الزمن الحقيقي المخصصة

ظهرت واختفت العديد من لغات الزمن الحقيقي المتخصصة على مدى الثلاثين سنة الماضية. ونقدم فيما يلي بعضاً منها:

- **اللغة PEARL:** طُورت هذه اللغة من قبل مجموعة من الباحثين الألمان في بدايات السبعينيات، ولها تطبيقات واسعة خصوصاً في مجال الأتمتة الصناعية في ألمانيا.
- **لغة Real-Time Euclid:** وهي لغة تجريبية تتمتع بكونها من اللغات القليلة المناسبة تماماً لتحليل قابلية الجدولة **scheduability analysis**. للأسف، لم تجد هذه اللغة طريقاً لتحقيق تطبيقات عملية في مجال نظم الزمن الحقيقي.
- **لغة Real-Time C:** وهو اسم عام للعديد من التمديدات الماكروية للغة **C** التي تقدم بنى خاصة للتوقيت والتحكم الغير متوفرة في لغة **C** الأصلية.
- **لغة Real-Time C++:** وهو اسم عام للعديد من المكتبات الغرضية المطورة خصيصاً للغة **C++**، وهي تقدم إمكانات متقدمة للتوقيت والحكم.

جامعة دمشق

كلية الهندسة المعلوماتية

قسم النظم و الشبكات الحاسوبية

السنة الخامسة

Real Time System

Barber Shop

إعداد :

مصطفى محمد نجم

Moustafa-MN@hotmail.com

2008-2009

مقدمة :

يعتبر التزامن من أهم المشاكل التي تواجه المبرمج أثناء تنفيذ البرامج المختلفة، و بخاصة التي تشترك فيها أكثر من مهمة بموارد محددة.

هناك لغات برمجة صممت لتطبيقات نظم الزمن الحقيقي ،حيث أن تحوي اللغة في صلبها منطق المزامنة و قيود على الوصول إلى الموارد،و في هذه اللغات يقتصر دور المبرمج على توصيف العمل المطلوب و يترك مهمة المزامنة للغة بما تملكه من إمكانيات ،و لكن على المبرمج توصيف العمل بشكل دقيق .

من هذه اللغات لغة ADA التي سنستخدمها في برمجة حل لمسألة الحلاق BarberShop .

مقدمة و تاريخ مختصر عن ADA :

لغة Ada هي لغة عالية المستوى صممت من أجل برمجة الأنظمة ذات الزمن الحقيقي، و على مستوى عال و ضخم.

الاسم ADA مشتق من أوغستا ADA بيرن ابنة الأديب لورد بيرن، و عرفت بأول مبرمجة في العالم.

اخترعت ADA نتيجة تصور وكالة الدفاع الأمريكية أنه لا توجد لغة مناسبة جدا لتطبيقات أنظمة الزمن الحقيقي لنظم التحكم و الأنظمة Embedded Systems

Embedded Systems: التي هي عبارة عن نظام كمبيوتر موجود داخل نظام ما مثل الفرن الذكي أو الصواريخ الموجهة.

❖ مسألة الحلاق : Barber Shop Problem

توصيف المسألة :

نريد تمثيل عمل صالون الحلاق ،هذا الصالون يحوي عدد من الحلاقين N Barbers و كذلك عدد مساوي من كراسي الحلاقة N barber's chair ،و هناك غرفة انتظار تحوي عدد من الكراسي M waiting chairs.

- إذا لم يكن هناك زبائن ينام جميع الحلاقين،ريثما يأتي زبون.
- إذا كان جميع الحلاقين مشغولين ،و كان هناك كراسي شاغرة في غرفة الانتظار فيمكن للزبون أن يجلس على أحد الكراسي و ينتظر.
- إذا كان الحلاق نائم و جاء زبون ،فإن الزبون يقوم بإيقاظه.
- إذا انتهى أحد الحلاقين من الحلاقة لزبون فإنه يقوم بإيقاظ أحد الزبائن المنتظرين في غرفة الانتظار.و إذا لم يجد أي زبون فإنه يخلد للنوم،و يستغرق في الأحلام.
- إذا جاء زبون و وجد غرفة الانتظار ممتلئة و جميع الكراسي مشغولة ،فإن الزبون ينتظر لفترة على باب المحل T ثانية،خلال هذه الفترة إذا لم يفرغ أي كرسي فإنه يغادر المحل (الرجاء العودة في وقت لاحق..!!).

إذا من معطيات المسألة يمكن أن نلخص النقاط الأساسية :

- N barber , M customer .
- لدينا فترة انتظار للزبون T second .

ملاحظة :

يمكن أن نعتبر جميع الكراسي في المحل هي كراسي انتظار و لا نفرق بين كرسي انتظار و كرسي حلاق لأنها من حيث المنطق جميعها راسي موجودة داخل المحل.

و قد لاحظت ذلك من خلال الحل.

بنى المعطيات المستخدمة :

نستخدم في الحل مايلي :

- **الوحدة المحمية Protected Unit :**

– إن الوحدة المحمية تشبه إلى حد ما مفهوم التغليف في لغات البرمجة التقليدية .

– تحوي الموارد المشتركة (التي نحتاج إلى تطبيق التزامن عليها و منع الوصول إليها من قبل أكثر من مهمة بنفس الوقت) ،و التوابع و الإجرائيات التي تتعامل معها.

يمكن هنا أن نستخدم ثلاثة أنواع من التعريفات:

Function : للقراءة فقط ،لا يمكنه التعديل ،و هو يعيد قيمة.

Procedure :للكتابة ،يمكنه التعديل على المتحولات .

Entry : و هو عبارة عن procedure مع شرط (لا يتم الدخول إليها إلا عند تحقق الشرط).

كما نعرف المتحولات ضمن نطاق Private .

– نعرف الوحدة المحمية : **barbershop**

```
protected barbershop is
    entry take_chair;
    procedure leave;
    function howmany return integer;
    private
    num: integer:=0;
   chair_Max : integer:=5;
end barbershop;
```

- المتحولات : Private

  - **num** : عدد الزبائن في غرفة الانتظار ،و قيمته الابتدائية = 0 .
  - **chair_Max** :عدد الكراسي في غرفة الانتظار ،و هو عدد ثابت.

- التوابع و الإجرائيات:

- entry take_chair when (num<chair_Max)

و هو عبارة عن Entry يقوم بزيادة عدد المنتظرين ،يطلبه الزبون عندما يأتي إلى المحل ليحجز كرسي .

و لا يتم الدخول إليه إلا إذا كان عدد المنتظرين أقل من عدد الكراسي المتاحة (هناك كراسي شاغرة)

- function howmany return integer

و هو عبارة عن تابع يعيد عدد المنتظرين في غرفة الانتظار (يقابل Getter في لغات البرمجة التقليدية و الذي نستخدمه للحصول على قيمة متحول خاص private).

- procedure leave

و هي عبارة عن إجرائية يستدعيها الزبون عندما يغادر ،و تقوم بإنقاص عدد المنتظرين بمقدار 1 .

• المهمة Task :

- المهمة هي الوحدة التنفيذية في Ada و يبدأ تنفيذها من نقطة تعريفها (أو لحظة خلقها أي لا تحتاج إلى تشغيل من قبل المبرمج كما في النيسب Thread و الذي يتم تشغيله من خلال Start)
- بما أنه لدينا عدة زبائن و عدة حلاقين لذلك نستخدم Task Type والتي تعرف لنا نمط مهمة ثم نخلق مهام من هذا النمط.
- سنعرف نمط مهمة للحلاق و نمط مهمة للزبون.

- **مهمة الحلاق : barber**

```
task type barber(id:integer) is
   entry haircut;
end barber;
```

تحوي هذا المهمة Entry واحد و هو يمثل عملية الحلاقة حيث نعرف Accept له في جسم المهمة و داخلها نقوم بعملية انتظار فقط (تمثل فترة الحلاقة).

تأخذ هذه المهمة كدخل رقم مميز (فقط لتمييزها أثناء الطباعة).

- **مهمة الزبون : customer**

الزبون إما يدخل و يجلس على أحد الكراسي (في حال كان هناك كراسي شاغرة) و يطلب أحد الحلاقين (من خلال عملية select).

أو في حال كانت غرفة الانتظار ممتلئة : فإنه ينتظر لفترة زمنية ،و في حال لم يفرغ كرسي فإنه يغادر دون حلاقة.

- نستخدم هنا مفهوم **Rendezvous** للمزامنة بين الإجرائيات.

Rendezvous:

و هي طريقة للمزامنة بين مهمتين هما : المستدعي caller و المستدعى callee.

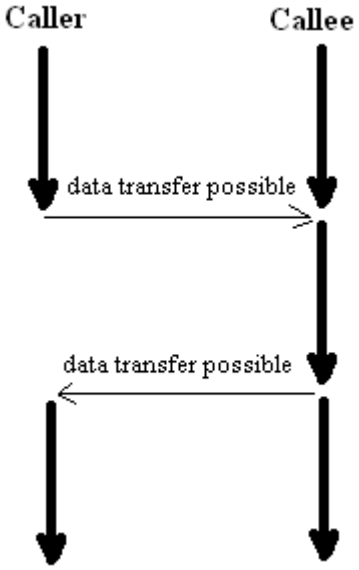
المستدعي هنا هو الزبون و المستدعى هو الحلاق.

حيث نعرف entry في المستدعى تمثل خدمة يقدمها للزبائن (و هي الحلاقة هنا haircut).

المستدعى ينتظر إرسال طلب من أحد الزبائن ليقوم بقبول الطلب و تخديمه (المستدعى ينتظر عند العبارة التي تمثل قبول الطلب و ينام ما لم يكن هناك طلبات).

المستدعي بدوره يرسل طلب و ينتظر حتى يتم تخديمه (أي يصف بالدور و ينام ثم يقوم المستدعى بإيقاظه عندما يحين دوره).

يمكن تمثيله بالشكل التالي :



أي يرسل المستدعي الطلب و ينتظر ريثما ينتهي

المخدم من تخديمه(عندئذ يمكنه متابعة عمله).

هذا ضروري حتى لا نسمح للزبون بالذهاب

قبل أن تتم عملية حلاقة شعره.

سيناريو العمل:

- **الزبون :**

– يأتي الزبون فيطلب take_chair (هنا نضمن أنه وحده من يعدل على num لأنها موجودة ضمن PU)

– إما يجد مكان فيدخل و يجلس و ينتظر أحد الحلاقين حتى يحلق له.

– أو لا يجد مكان فينتظر على الباب فترة زمنية علها تفرغ أحد الكراسي ضمن هذه الفترة .

- إذا فرغت أحد الكراسي خلال فترة الانتظار ،يدخل الزبون المنتظر ،و بالتالي لم يذهب الانتظار عالفاضي
- إذا انتهت فترة الانتظار و لم يفرغ أي كرسي ،يغادر الزبون المنتظر ،دون حلاقة ..!!

– عندما يدخل الزبون يقوم بإيقاظ الحلاق (في حال كان نائم) من خلال إرسال طلب.

– أو يقوم بالانتظار على أحد الحلاقين و ينام ،و الحلاق بدوره يوقظه من خلال عملية Accept.

- **الحلاق:**

– ينفذ إجرائية تخديم الطلبات باستمرار:

– فإذا لم يكن هناك طلبات (لا يوجد زبائن) فإنه ينام (يتوقف عند هذه النقطة حتى لحظة وصول طلب).

(عندما يأتي زبون فإنه يوقظ الحلاق من خلال إرسال طلب .. الحلاق يكون مسبقا منتظرا هذه اللحظة).

– أو يقوم بإيقاظ أحد الزبائن النائمين (المنتظرين) ليحلق له شعره.

– يقوم الحلاق بتخديم الطلب (حلاقة شعر الزبون).

– ثم يكرر العملية ليأخذ زبون آخر (في حال تواجده).

خوارزمية الحل :

Customer	Barber

Task type customer;
Task body customer is
select
Barbershop.take_chair;
select
Barber1.haircut;
Barbershop.leave;
or
Barber2.haircut;
Barbershop.leave;
or
Barber3.haircut;
Barbershop.leave;
or
delay T;
end customer;
	Task type barber is
	entry haircut;
	end barber ;
	Task type barber is
	loop
	If (barbershop.howmany=0) then
	put_line(“sleeeeep”);
	end if;
	accept haircut do
	delay (haircut_time);
	end haircut;
	end loop;
	end barber;

✓ **ملاحظة** : يتم تحرير الكرسي بعد الحلاقة لأننا نعتبر أن كراسي الحلاقة هي من كراسي الانتظار و لا نفرق بينها ،و بالتالي في حال كانت الغرفة ممتلئة لا يمكن دخول زبون جديد إلا بعد انتهاء أحد الزبائن من الحلاقة و مغادرته المحل و هذا منطقي. (ما يحدث في الواقع).

الحل باستخدام **Rendezvous** :

```
-- Barbershop program Using Ada (Rendezvous)
--Author Moustafa Najm ..... M.N : Moustafa-MN@hotmail.com
with Ada.Text_IO;
use Ada.Text_IO;

procedure Barber_Main is
  -------------------------------- Protected Unit barbershop ---------------------------
  --barbershop declaration
  protected barbershop is
    entry take_chair;
    procedure leave;
    function howmany return integer;

    private
    num: integer:=0;
    chair_Max : integer:=5;
  end barbershop;

  --barbershop body
  protected body barbershop is

    entry take_chair when (num<chair_Max)is
    begin
      num :=num+1;
      put_Line("customer take chair");
    end take_chair;

    function howmany return integer is
    begin
      return num;
    end howmany;

    procedure leave is
    begin
      num :=num-1;
      put_line("customer left ..number of customer waiting :"&num'img);
    end leave;

  end barbershop;
```

```
 ------------------------------------ Barber Task Type ------------------------------------
task type barber(id:integer) is
  entry haircut;
end barber;
task body barber is
begin
  loop
    if(barbershop.howmany=0)then
      put_line("barber"&id'img&" : I'm going to sleep");
    end if;
    accept haircut do
        put_line("barber"&id'img&" is cutting hair");
        delay 2.0;
      end haircut;
  end loop;
end barber;

---------------------------------- Barbers Instantiation -------------------------------
barber1:barber(1);
barber2:barber(2);
barber3:barber(3);

---------------------------------- Customer Task Type ----------------------------------
task type customer (id:integer);
task body customer is
begin
  delay 1.0;
  put_Line("customer"&id'img&"  arrive to barber shop");
  select
    barbershop.take_chair;
    put_Line("customer"&id'img&" enter");
    select
      barber1.haircut;
    else
      select
        barber2.haircut;
      else
        barber3.haircut;
      end select;
    end select;
    barbershop.leave;
  or
```

```
      delay 3.0;
      put_Line("I wait for 3 second...!! customer"&id'img&" will leave");
    end select;

  end customer;
    ------------------------------------ Customers Instantiation -------------------------
  customer1:customer(1);
  customer2:customer(2);
  customer3:customer(3);
  customer4:customer(4);
  customer5:customer(5);
  customer6:customer(6);
  customer7:customer(7);
  customer8:customer(8);
  customer9:customer(9);
  customer10:customer(10);

begin
  null;
end;
  ---------------------------------------- End Program... M.N ----------------------------
```

ملاحظات:

- **المهمة تبدأ التنفيذ اعتبارا من لحظة خلقها .. و لذلك نجد إجرائية البرنامج الرئيسي فارغة.**
- يمكن أن نضع كذلك فترة انتظار عظمى للحلاق .. مثلا إذا لم يأت أي زبون خلال 10 ثواني ..!!

  فإنه يغلق المحل و يذهب للبيت .. أو بروح مشوار.

```
Select
        accept  haircut  do
              ……..
          end haircut;
      or delay 10.0;
end select;
```

M.N MMNajm@Gmail.com

التنفيذ : 3 barbers , 10 customers , 5 Chairs

```
C:\Documents and Settings\M.N>gnatmake barber_main.adb -o barber
gnatbind -x barber_main.ali
gnatlink barber_main.ali -o barber.exe

C:\Documents and Settings\M.N>barber

barber 1 : I'm going to sleep
barber 2 : I'm going to sleep
barber 3 : I'm going to sleep
customer 4  arrive to barber shop
customer take chair
customer 4 enter
barber 1 is cutting hair
customer 5  arrive to barber shop
customer take chair
customer 5 enter
barber 2 is cutting hair
customer 6  arrive to barber shop
customer take chair
customer 6 enter
barber 3 is cutting hair
customer 7  arrive to barber shop
customer take chair
customer 7 enter
customer 1  arrive to barber shop
customer take chair
customer 1 enter
customer 2  arrive to barber shop
customer 3  arrive to barber shop
customer 10  arrive to barber shop
customer 8  arrive to barber shop
customer 9  arrive to barber shop
customer left ..number of customer waiting : 4
customer take chair
customer 2 enter
customer left ..number of customer waiting : 4
customer take chair
customer 3 enter
barber 3 is cutting hair
barber 1 is cutting hair
customer left ..number of customer waiting : 4
customer take chair
customer 10 enter
chairs are full ..I wait for 3 second .. customer 8 will leave
chairs are full ..I wait for 3 second .. customer 9 will leave
```

customer left ..number of customer waiting : 4
customer left ..number of customer waiting : 3
barber 3 is cutting hair
customer left ..number of customer waiting : 2
barber 3 is cutting hair
customer left ..number of customer waiting : 1
barber 3 is cutting hair
customer left ..number of customer waiting : 0
barber 3 : I'm going to sleep

تعليق على التنفيذ:

- **في البداية .. جميع الحلاقين يغطون في نوم عميق (لا يوجد زبائن ..!!).**
- **يصل أول خمسة زبائن 4,5,6,7,1 : يدخلون مباشرة ،و يجلسون على كراسي و يوقظون الحلاقين و يبدأ الحلاقون بالحلاقة لهم.**
- **نلاحظ بعدها يصل 2,3,10,8,9 ينتظرون على الباب لعدم وجود كراسي فارغة.**
- **ينتهي أحد الزبائن من الحلاقة و يغادر فيدخل 2 .**
- **ثم ينتهي زبون آخر من الحلاقة فيدخل 3.**
- **ثم ينتهي زبون آخر من الحلاقة فيدخل 10.**
- **لاحظ تغير عدد الزبائن المنتظرين و دخول أحد المنتظرين على الباب عند مغادرة كل زبون.**
- **8 و 9 ينتظرون فترة 3 ثواني على الباب و لا يستطيعون الدخول فيغادرون دون حلاقة.**
- **أخيرا ينتهي جميع الزبائن من الحلاقة و يغادرون،و يعود الحلاقون إلى نومهم و يستغرقون في أحلامهم**

ملاحظة :نلاحظ أنه كلما زدنا مدة انتظار الزبائن على الباب يقل عدد الزبائن الذين يغادرون دون حلاقة ،و يظهر ذلك جليا من خلال تغيير المدة و التنفيذ.

M.N Moustafa-MN@hotmail.com

الحل باستخدام **Requeue** :

```
-- Barber program
--this solution for Barberer problem by Ada with PU & Requeue
--Author Moustafa Najm ..... M.N : Moustafa-MN@hotmail.com

with Ada.Text_IO;
use Ada.Text_IO;

procedure main is

  ---------------------- Protected Unit Shop ----------------------
  protected shop is
    entry enter;
    procedure leave;

  private
    count:integer:=0;
    barber:integer:=3;
    Max:integer:=5;
    entry share;
  end shop;

  protected body shop is

    entry enter when count<Max is
    begin
      count:=count+1;
      put_line("custumer enter .. and wait now to shar");
      requeue share;
    end enter;

    entry share when barber>0 is
    begin
      barber:= barber-1;-- get haircut
    end share;

    procedure leave is
    begin
      count :=count-1;
      barber:=barber+1;
    end leave;

  end shop;
```

```
 ----------Barber Task ----------
task  Barber; -- there is no thing to do .. we can remove
task body barber is
begin
  null;
end;
---------- Customer Task Type ----------
task Type customer(id:integer);
task body customer is
begin
  select
    shop.enter;
    put_Line("customer"&id'img&" enter .. and he is share now");
    delay 2.0;
    shop.leave;
    put_Line("customer"&id'img&" shared and left");
  or
    delay 3.0;
    put_Line("I wait for 3 second...!! customer"&id'img&" will leave");
  end select;
end customer;
---------- Customers Instantiation ----------
customer1:customer(1);
customer2:customer(2);
customer3:customer(3);
customer4:customer(4);
customer5:customer(5);
customer6:customer(6);
customer7:customer(7);
customer8:customer(8);
customer9:customer(9);
customer10:customer(10);

begin
  null;
end main;
  ---------- End Main ... M.N ----------
```

التنفيذ :

```
custumer enter .. and wait now to shar
customer 1 enter .. and he is share now
custumer enter .. and wait now to shar
customer 2 enter .. and he is share now
custumer enter .. and wait now to shar
customer 3 enter .. and he is share now
```

```
custumer enter .. and wait now to shar
custumer enter .. and wait now to shar
customer left ..number of customer waiting : 4
custumer enter .. and wait now to shar
customer 4 enter .. and he is share now
customer 1 shared and left
customer left ..number of customer waiting : 4
custumer enter .. and wait now to shar
customer 5 enter .. and he is share now
customer 2 shared and left
customer left ..number of customer waiting : 4
custumer enter .. and wait now to shar
customer 6 enter .. and he is share now
customer 3 shared and left
I wait for 3 second...!! customer 9 will leave
I wait for 3 second...!! customer 10 will leave
customer left ..number of customer waiting : 4
customer 7 enter .. and he is share now
customer 4 shared and left
customer left ..number of customer waiting : 3
customer 8 enter .. and he is share now
customer 6 shared and left
customer left ..number of customer waiting : 2
customer 5 shared and left
customer left ..number of customer waiting : 1
customer 8 shared and left
customer left ..number of customer waiting : 0
customer 7 shared and left
```

ملاحظة :

لاحظ هنا أن الحلاق Barber ليس له أي أهمية ،حتى أنه يمكن حذف المهمة Barber بشكل كامل.

(حيث يهمنا فقط عدد الحلاقين ،و هو عبارة عن متحول موجود ضمن الوحدة المحمية).

لا تحاول أن تجعل ملابسك أغلى شيء فيك ، حتى لا تجد

نفسك يوماً أرخص مما تريديه

الحل باستخدام Semaphore

```
-- Barbershop program Using Ada (Semaphore)
--Author Moustafa Najm ... M.N : Moustafa-MN@hotmail.com

with Ada.Text_IO;
use Ada.Text_IO;

procedure barbershop is

   protected type semaphore is

 procedure decrement ;
 procedure up;
 entry down;
 private value: integer:=1;
   end semaphore;

   protected body semaphore is
      procedure decrement is
      begin
         value:= value+1;
      end decrement;
      procedure up is
      begin
         value:=value+1;
      end up;
      entry down when value>0 is
      begin
         value:=value-1;
      end down;
   end semaphore;

   mutex , barber , customer : semaphore;
   customers: integer:=0;
   max :constant integer:=5;

   task type barberTask;

   task body barberTask is
   begin
      customer.decrement;
      barber.decrement;
      loop
         customer.down;
         delay 2.0;
         barber.up;
      end loop;
   end barberTask;

   task type customerTask (id:integer);
   task body customerTask is
   begin
      --we use loop to round customers and generate infinite customers
      loop
         mutex.down;
      if customers = max then
```

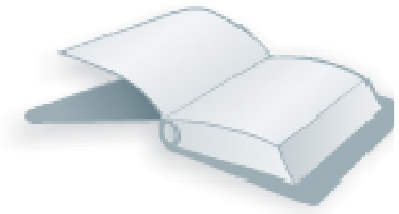
```
         mutex.up;
         put_line("waiting room is full customer"&id'img&" will leave");

      else
         customers:=customers+1;
         mutex.up;
         customer.up;
         barber.down;
         mutex.down;
         customers:=customers-1;
         mutex.up;
         put_line("Customer "& id'img &" is getting a hair-cut");
      end if;
         delay 2.0;
      end loop;
   end customerTask;

   bar1: barberTask;
   bar2: barberTask;
   bar3: barberTask;

   customer1:customerTask(1);
   customer2:customerTask(2);
   customer3:customerTask(3);
   customer4:customerTask(4);
   customer5:customerTask(5);
   customer6:customerTask(6);
   customer7:customerTask(7);
   customer8:customerTask(8);
   customer9:customerTask(9);
   customer10:customerTask(10);

begin
   null;
end barbershop;
```

التنفيذ:

```
   Customer  1 is getting a hair-cut
   Customer  2 is getting a hair-cut
   Customer  3 is getting a hair-cut
   Customer  4 is getting a hair-cut
   waiting room is full customer 10 will leave
   Customer  5 is getting a hair-cut
   Customer  6 is getting a hair-cut
   Customer  7 is getting a hair-cut
   waiting room is full customer 2 will leave
   waiting room is full customer 10 will leave
   waiting room is full customer 5 will leave
   waiting room is full customer 6 will leave
   waiting room is full customer 7 will leave
   Customer  8 is getting a hair-cut
   Customer  9 is getting a hair-cut
   Customer  3 is getting a hair-cut
   Customer  4 is getting a hair-cut
```

الحل باستخدام Package

```
-- Barbershop program Using Ada (Package)
-- Author Moustafa Najm ..... M.N : Moustafa-MN@hotmail.com

-- This file contains the interface definition for the Barbershop
-- Barbershop.Ads
generic
   Num_Chairs : Positive;

package BarberShop is

   subtype Customer is Positive;
   type Barber_Queue is array(Positive range 1..Num_Chairs)
                     of Customer;

   protected Barber_Chairs is

      function Is_Full return Boolean;
      entry Take_Chair(The_Customer : in Customer);
      entry Wait_For_Customer(The_Customer : out Customer);

   private
      Queue : Barber_Queue;
      Next_Available : Positive := 1;
      Next_Customer : Positive := 1;
      Num_Waiting : Natural := 0;
   end Barber_Chairs;

   task Barber is
      entry Stop;
   end Barber;

   task Demo_Master;

end BarberShop;
```



```
-- This file contains the implementation of the Barbershop
-- BarberShop.adb

with Ada.Text_Io;
use Ada.Text_Io;
   ---------------------- Package Body BarberShop --------------------------
package body BarberShop is

   --------------------- Protected Unit barber_chairs --------------------
   protected body Barber_Chairs is

      --------------------function Is_Full -----------------------
      function Is_Full return Boolean is
      begin
         return Num_Waiting = Num_Chairs;
      end Is_Full;

      --------------------Entry Take_chair -----------------------
      entry Take_Chair (The_Customer : in Customer ) when not Is_Full is
      begin
         Queue(Next_Available) := The_Customer;
         if Next_Available = Num_Chairs then
            Next_Available := 1;
         else
            Next_Available := Next_Available + 1;
         end if;
         Num_Waiting := Num_Waiting + 1;
      end Take_Chair;

      --------------------Entry Wait_For_Costomer ---------------------
      entry Wait_For_Customer ( The_Customer : out Customer ) when Num_Waiting >
0 is
      begin
         The_Customer := Queue(Next_Customer);
         if Next_Customer = Num_Chairs then
            Next_Customer := 1;
         else
            Next_Customer := Next_Customer + 1;
         end if;

      end Wait_For_Customer;

   end Barber_Chairs;

      ------------------------- Task Type Barber --------------------------
   task body Barber is
      Current_Customer : Customer;

   begin
      loop
```

```
         select
            Barber_Chairs.Wait_For_Customer(Current_Customer);
            Put_Line("Serving customer" & Customer'Image(Current_Customer));
            delay 2.0; -- a 2 second hair cut
         else
            delay 1.0; -- sleep one second waiting for a customer
            select
               accept Stop;
               exit;
            else
               delay 0.0;
                  Put_Line("ZZZZZZZZZZ");
            end select;
         end select;
      end loop;

   end Barber;

      ------------------------- Task Demo_Master ----------------------------
   task body Demo_Master is
      Cust_Max : constant Customer := 20;

   begin
      delay 2.0;
      for Cust in 1..Cust_Max loop
         delay 0.5;
         if not Barber_Chairs.Is_Full then
            Barber_Chairs.Take_Chair(Cust);
            Put_Line("Customer" & Customer'Image(Cust) &" took a chair
                     in the barber shop.");
         else
            Put_Line("Shop full, customer" & Customer'Image(Cust) &"left shop");
         end if;
      end loop;
      delay 3.0;
      Barber.Stop;
   end Demo_Master;
end BarberShop;

   ----------------------End Package Body BarberShop ----------------------
-- This file contains Main Procedure
-- Barber_Main.adb

with Barbershop;
procedure Barber_Main is
   package Small_Shop is new BarberShop(Num_Chairs => 5);

begin
   null;
end Barber_Main;
```

التنفيذ :

```
ZZZZZZZZZZ
ZZZZZZZZZZ
Customer 1 took a chair in the barber shop.
Customer 2 took a chair in the barber shop.
ZZZZZZZZZZ
Serving customer 1
Customer 3 took a chair in the barber shop.
Customer 4 took a chair in the barber shop.
Customer 5 took a chair in the barber shop.
Serving customer 2
Customer 6 took a chair in the barber shop.
Customer 7 took a chair in the barber shop.
Shop full, customer 8 left shop
Shop full, customer 9 left shop
Serving customer 3
Customer 10 took a chair in the barber shop.
Shop full, customer 11 left shop
Shop full, customer 12 left shop
Shop full, customer 13 left shop
Serving customer 4
Customer 14 took a chair in the barber shop.
Shop full, customer 15 left shop
Shop full, customer 16 left shop
Shop full, customer 17 left shop
Serving customer 5
Customer 18 took a chair in the barber shop.
Shop full, customer 19 left shop
Shop full, customer 20 left shop
Serving customer 6
Serving customer 7
Serving customer 10
Serving customer 14
Serving customer 18
```

تم بعونه تعالى ....

Moustafa Najm **Moustafa-MN@hotmail.com**